Regression Test Selection by Exclusion

Amir Ngah

School of Engineering and Computing Sciences

Durham University

A thesis submitted for the degree of

Doctor of Philosophy

May 2012

Acknowledgements

First of all, all praise and thanks go to Allah for all the success in this thesis in particular and in my life in general.

Secondly, my deep thanks to my supervisor Emeritus Professor Malcolm Munro for his invaluable advice and guidance. Professor Malcolm gave me all benefits from his wide experience. My thesis would not be completed and succeeded without his great supervision. Thanks also to my former supervisor Dr Keith Gallagher that gave me a chance for doing my PhD study in Durham University, my Internal Examiner Dr Ioannis Ivrissimtzis and External Examiner Dr Steve Counsell for giving invaluable comments and suggestions to improve my thesis.

Thirdly, my sincere thanks to my great father (Hj Ngah Hj Ali) and mother (Hjh Zainun Hj Mat Teh) for their constant educate and advice me since I was born. My special thanks to my beloving wife Nik Marlina Nik Mohd for her constant encouragement, support, prays, and patience in what ever situation. Also my thanks to my brothers and sisters, other family members, office mate (Mohammad, Lutfi, Daniel, Khalid, Maria) and all my friends for their support and pray.

Abstract

This thesis addresses the research in the area of regression testing. Software systems change and evolve over time. Each time a system is changed regression tests have to be run to validate these changes. An important issue in regression testing is how to minimise reuse the existing test cases of original program for modified program. One of the techniques to tackle this issue is called regression test selection technique. The aim of this research is to significantly reduce the number of test cases that need to be run after changes have been made. Specifically, this thesis focuses on developing a model for regression test selection using the decomposition slicing technique.

Decomposition slicing provides a technique that is capable of identifying the unchanged parts of the system. The model of regression test selection based on decomposition slicing and exclusion of test cases was developed in this thesis. The model is called Regression Test Selection by Exclusion (ReTSE) and has four main phases. They are Program Analysis, Comparison, Exclusion and Optimisation phases.

The validity of the ReTSE model is explored through the application of a number of case studies. The case studies tackle all types of modification such as change, delete and add statements. The case studies have covered a single and combination types of modification at a time. The application of the proposed model has shown that significant reductions in the number of test cases can be achieved. The evaluation of the model based on an existing framework and comparison with another model also has shown promising results. The case studies have limited themselves to relatively small programs and the next step is to apply the model to larger systems with more complex changes to ascertain if it scales up. While some parts of the model have been automated tools will be required for the rest when carrying out the larger case studies.

Copyright

Declaration

The material contained within this thesis has not previously been submitted for a degree at the Durham University or any other university. The following publication was produced during the course of this thesis:

- Amir Ngah and Keith Gallagher, "Regression test selection by exclusion using decomposition slicing", *In Proceedings of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, Amsterdam, The Netherlands, 2009, pp. 23-24.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Research Overview

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software [1]. It is also defined as a systematic approach to the analysis, design, assessment, implementation, testing, maintenance and reengineering of software [70]. The generic activities in all software processes are requirement analysis, design, coding, testing and maintenance [82; 97].

Software testing is an important activity in software development. It identifies defects and problems, and evaluates and improves product quality. Software testing has been a serious research topic since the late 1960s [56; 87]. Software testing may represent more than 40% of a software development budget [13; 48; 56; 64; 95]. Moreover, approximately 50% of the elapsed time is expended in testing software being developed [64; 65].

Software maintenance refers to the modifications of software after delivery. Other terms suggested for maintenance are *software support*, *software renovation*, *continuation engineering* and *software evolution* [11]. The IEEE Standard 1219-1993 [2] has defined software maintenance as the modification of a software product after delivery to correct faults, to improve performance or other

attributes, or to adapt the product to a modified environment. ISO/IEC 14764-1999 [4] has defined software maintenance as software product undergoes modification to code and associated documentation due to a problem or the need for improvement. Software systems change and evolve over time. It is impossible to develop any software which does not need to be modified. Each time a system is modified, regression tests have to be run to validate these modifications. This issue is called regression testing.

Regression testing is expensive but an essential activity in software maintenance. Regression testing attempts to validate modified software and ensures that the modified parts of the program do not introduce unexpected errors. The time used for regression testing can be assumed approximately half of the software maintenance activities. Improvements in the regression testing process will help to lower the elapsed time and the expenses of making changes to software. This thesis addresses the research in the area of regression testing.

There are a number of common terminologies that have to be defined in order to discuss regression testing. A *Certified* program refers to the previously tested version of a program. A *Modified* program refers to a program that is obtained by modifying the *certified* program. A *test suite* is a set of test cases that is used to test a program. T is the test suite that is used to test the *certified* program and T' is the test suite that is used to test the *modified* program. Regression testing involves creating test suite T', to validate the *modified* program, and running T' as an input to the *modified* program [14].

An important issue in regression testing is how to minimise reuse of the existing test cases of the *certified* program for the *modified* program. One of the techniques to tackle this issue is called regression test selection technique. There are many studies have discussed regression test selection techniques [10; 21; 22; 28; 43; 44; 51; 53; 54; 55; 89; 90; 102; 103; 110]. These techniques attempt to reduce the cost of regression testing by selecting appropriate test cases using information from the *certified* program, the *modified* program and the existing test suite. These techniques are classified as inclusion techniques because they select

test cases from the test suite.

The overall aim of this thesis is to discuss the research of regression test selection. Specifically, this research aims to reduce the number of test cases that need to be run after changes have been made. It focuses on developing a model for regression test selection by exclusion using the decomposition slicing technique. Exclusion means that the model excludes test cases that are not needed in regression testing. The decomposition slicing technique is capable of identifying the unchanged parts of the system. The model of regression test selection based on decomposition slicing and exclusion of test cases was developed in this thesis and called Regression Test Selection by Exclusion (ReTSE). The ReTSE model has four main phases: Program Analysis, Comparison, Exclusion and Optimisation. The initial paper of this model was published in the doctoral symposium for ESEC/FSE 2009 [77].

Rothermel and Harrold [88] proposed a framework for evaluating regression test selection techniques. It is based on four categories which are inclusiveness, precision, efficiency and generality. This framework is used to evaluate the ReTSE model based on some case studies. The case studies consider every types of modifications such as change, add and delete statements in the program. Then, the model is compared to the existing regression test selection technique.

1.2 Criteria for Success

There are four main criteria for success of this research. They are developing, implementing, analysing and evaluating the new regression test selection model.

1. Development of a new regression test selection model
   This research aims to develop a new regression test selection model. The model will select test cases from the existing test suite for the modified program. The research will use an exclusion mechanism and decomposition slicing technique in the proposed model.

2. Implementation of new model

The proposed model will be implemented through a development of a prototype. The research also will discuss existing tools that are suitable for the proposed model.

3. Analysis of new model
   The proposed model will be analysed based on a number of case studies. The case studies use small programs that represent all types of modification such as change, add and delete statements.

4. Evaluation of new model
   The evaluation of the proposed model will be divided into two parts. The first part is based on the existing evaluation framework developed by Rothermel and Harrold [88]. The second is based on a comparison between the proposed model and other existing regression test selection techniques.

1.3 Thesis Outline

This thesis is divided into three main parts; background, the proposed model, and discussions. The background consists of Chapter 1 and Literature Review which is divided into three chapters. They are Software Testing (Chapter 2), Regression Testing (Chapter 3) and Program Slicing (Chapter 4). The proposed model includes Chapter 5, a description of the model, and Chapter 6, an implementation of the model. The discussions involves case studies in Chapter 7, analysis and evaluation in Chapter 8 and conclusions of the thesis in Chapter 9. The full structure of the thesis is as follows.

Chapter 2 provides the basic knowledge of software testing. The chapter provides some relevant definitions of software testing. Then, it discusses the different classifications of testing techniques and testing processes. Finally, the chapter discusses testing in software maintenance.

Chapter 3 provides a background of regression testing and begins with some definitions of regression testing. Then, it discusses regression testing strategies and categories and provides an evaluation framework for regression test selection

technique. The main focus is regression test selection techniques. Finally, the chapter discusses the implementation of regression testing in different environments.

Chapter 4 provides basic knowledge of program slicing because one of the existing slicing techniques is used in the proposed model. The chapter begins some definitions of program slicing. Then, it discusses representation types of programs or systems. This is followed by a discussion on various program slicing techniques, and in particular with specific reference to the decomposition slicing technique that is used in the proposed model. Finally, it describes the applications of program slicing.

Chapter 5 describes the proposed model called Regression Test Selection by Exclusion (ReTSE). The ReTSE model uses the decomposition slicing technique. Finally, the chapter uses a simple program to evaluate the model.

Chapter 6 presents the prototype of the ReTSE model. This chapter presents the existing tools used in the relevant phases in the ReTSE model. However, there are few phases that still work manually. Finally, the chapter discusses how to fully implement the ReTSE model in the future.

Chapter 7 discusses case studies that represent five types of modification. The five types of modification are change statements, add statements, delete statements, add variables and delete variables. Five case studies represent one type of modification at a time. Another two case studies have been presented to tackle a combination of modification types.

Chapter 8 discusses the analysis and evaluation of the ReTSE model in further details. The analysis is based on the case studies that are presented in Chapter 7. The evaluation is divided into two parts, the first is based on the existing framework for regression test selection techniques. The second is the comparison between the ReTSE model and the Pythia technique.

Chapter 9 provides the summary of the research. It also reviews the criteria for success that are presented in Chapter 1. This chapter concludes with suggestions to enhance the ReTSE model in future.

Chapter 2

Software Testing

2.1 Introduction

There are various definitions of software testing that are related to specific issues or problems. Some early definitions are from Myers [64] that define testing as the process of executing a program with the intent of finding faults. Hetzel [56] has defined testing as any activity aimed at evaluating an attribute or capability of a program or system. Testing is a measurement of software quality.

Software testing also known as a dynamic verification. The actual behaviour of a program on a set of test cases is compared to the expected behaviour [17]. Software testing is an iterative process [95], which consists of test designing, test execution, problems and problem repair, for validating functionality, and as well as attempting to break the software.

This chapter discusses software testing in general. The chapter is organised

as follows. The second section presents classifications of testing techniques. The third section presents testing processes. The fourth section discusses testing in maintenance processes.

2.2 Classifications of Testing Techniques

The classifications of testing techniques are divided into three parts. These are:

1. Static and dynamic testing [8; 81; 87].
2. Black-box and white-box testing [8; 87; 97].
3. Manual and automated testing [81; 95].

2.2.1 Static and Dynamic Testing

2.2.1.1 Static Testing

Static testing does not involve actual program execution. Usually, the developer who wrote the code uses this type of testing in isolation. Static testing is mostly used in requirements, design and coding phases. For instance, in static testing, specifications are compared with each other to verify that errors have not been introduced during the process. This comparison process is illustrated in Figure 2.1 [8]. The down arrow shows the translation process of information from the previous artifact to the next artifact. The up arrow shows the verification process of that translation. The artifacts are compared with each other to verify that each artifact accurately translates information from the previous artifact.

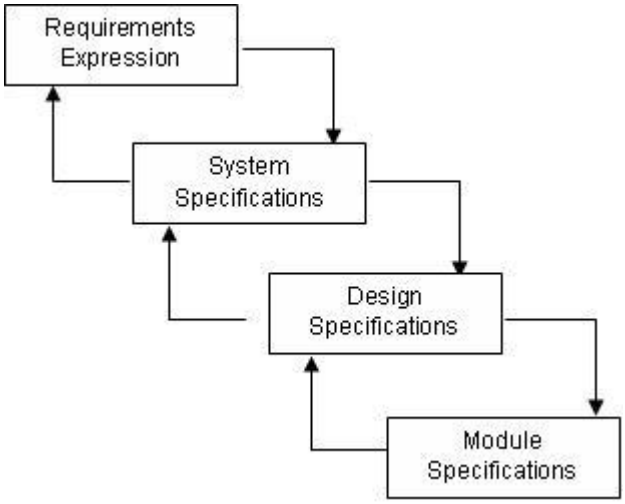

Figure 2.1: Using Specifications in Static Testing [8]

2.2.1.2 Dynamic Testing

Dynamic testing is a process of software execution on some test cases and examining the results to check whether it operated as expected [8]. It is also the process to confirm that the software functions according to its specification.

The test plan is the important aspect in dynamic testing [8]. It links the specification to the software to be tested. It should include a complete description of the strategy for testing, plus the test scripts and expected results. The results of dynamic tests are always compared with the expected results listed in the test plan. Any differences that are found must be resolved. After any necessary changes have been made, the tests will run again. This is part of the regression testing process that will be discussed in next chapter.

2.2.2 Black-box vs White-box Testing

2.2.2.1 Black Box Testing

Black-box testing assumes the software as a black box without any knowledge of internal implementation. Test cases derived from the program specification are called black-box techniques. In addition, black-box testing techniques are sometimes referred as functional or specification-based testing. The only information that is used in the functional approach is the specification of the program [66]. There are two distinct advantages of functional based testing. First, they are independent of how the program is implemented, so the test cases will not be effected if the implementation changes. Second, the development of test cases can occur in parallel with the implementation. This can reduce the overall project development time. On the other hand, functional test cases usually face two problems. Firstly, there can be significant redundancies amongst test cases. Secondly, some parts of the tested software may not be tested by functional test cases because the testers do not know the real code of that software.

2.2.2.2 White Box Testing

Test cases derived from a program itself are called white-box techniques. White-box testing is also called structural or code-based testing. There are two main white-box or structural testing techniques; control flow testing and data flow testing [101; 105]. The program shown in Figure 2.2 can be represented as a Control Flow Graph (CFG) as shown in Figure 2.3. Every statement in the program is represented by nodes. The flow from one node to another node is called an *edge*. Nodes 1 and 4 are called *predicate* nodes because they have more that one out

going edge. A *path* is the flow from the start node (node 1) to the end node (node 7). Nodes 6 and 7 are non-branching statements which can be treated as one statement unit [30]. There are four unique paths through the program in Figure 2.2. The paths are P1 ={1, 2, 4, 6, 7}, P2 = {1, 2, 4, 5, 6, 7}, P3 = {1, 3, 4, 6, 7} and P4 = {1, 3, 4, 5, 6, 7}.

```
1  if (condition1)
2     x = 1;
   else
3     x = 2;
4  if (condition2)
5     x = 10 * x;
6  y =  x + 10 / x;
7  write(x, y)
```

Figure 2.2: The Program for CFG

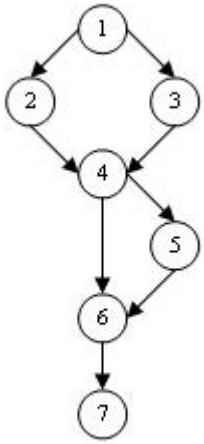

Figure 2.3: The Control Flow Graph

The effectiveness of structural testing is measured by *test coverage*, which measures the code exercised by the test cases [30]. Common coverage metrics

are statement, branch, and path coverage. Statement coverage requires that all statements in the program are executed at least one test case [30; 78; 87; 105]. The 100% statement coverage of the program in Figure 2.2 can be achieved by two test cases from the four paths. The combination of two test cases might be from {P1, P4}, {P2, P3}, or {P2, P4}. The suitable combination of test cases is important in statement coverage.

Branch coverage is also called decision coverage [30; 87]. It requires all branches of the program are traversed at least one test case. The 100% branch coverage of the program in Figure 2.2 also can be achieved by two test cases from the four paths. The combination options are {P1, P4} and {P2, P3}. Statement coverage and branch coverage can be achieved by exercising every path through the program which is called path coverage. Path coverage is a reliable technique, however, it is not possible for large systems because the number of paths is exponential with respect to the number of branches [30].

Data flow based testing basically uses definitions (*def*) and uses of variables (*use*) in the program [9; 36; 66; 85; 105]. A *def* is a statement that assigns a value to a variable or as an input of a variable. For instance, the statement 3 (x = 2) in Figure 2.2 is called *def* for the variable x. The occurrence of the variable x in the statement 6 (y = x + 10 / x) is called *use*. There are two types of *use*, namely computational use (*c-use*) and predicate use (*p-use*). The *c-use* is a statements where the value of a variable is used to compute the value of other variables or as an output value. The *p-use* is a statements where the value of variables is used in condition statements. The *def-use (du)* pair is a pair of definition of variable

and its uses which can be linked by a path without passing any other definition of the same variable.

2.2.3 Manual and Automated Testing

Manual software testing is the process of testing software that is carried out by an individual or group. Manual software testing uses more time and labour than automated testing. Automated software testing is a process of creating test scripts, which can then be run automatically, repetitively through several iterations. Automated software testing is more time efficient.

2.3 Testing Processes

Large systems should not be tested as a single entity. Large systems consist of sub-systems which are built out of modules which are composed of procedures and functions. The testing process starts from the small units and continues until the entire system is integrated. The most widely used testing process consists of five stages: unit testing, module testing, sub-system testing, system testing and acceptance testing [97]. In general, the main testing activities are component testing, integration testing and user testing. The sequence and relation between testing activities are shown in Figure 2.4.

Councill and Heineman [23] define a component as a software element that conforms to a standard components model and can be independently deployed

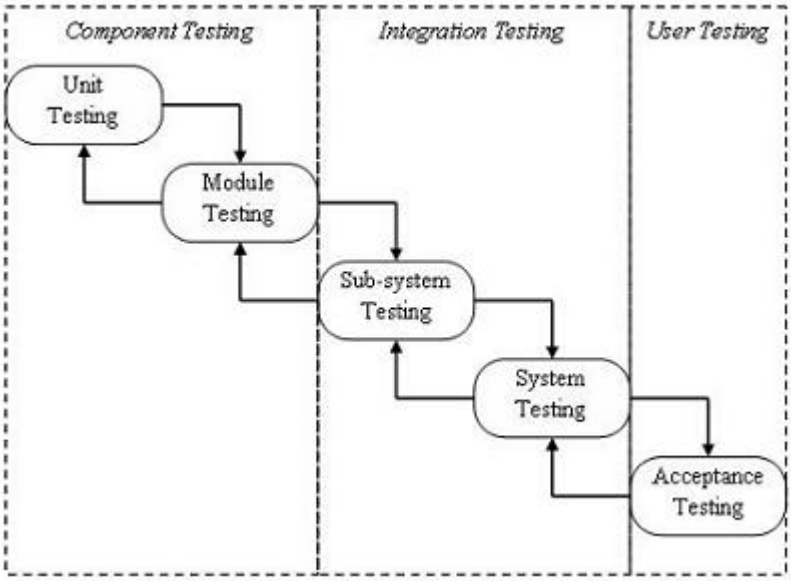

Figure 2.4: Testing Process in Software Development [97]

and composed without modification according to a composition standard. Component testing involves unit and module testing as shown in Figure 2.4. In unit testing, individual components are tested independently to ensure that they operate correctly. A module is a collection of dependent components that is tested in module testing process. Sub-system testing involves collections of modules which have been integrated into sub-systems. In the system testing, the system which consist of all integrated sub-systems is tested in order to validate that it meets its functional and non-functional requirements. Finally the acceptance testing will make sure the system is accepted for operational use. Acceptance testing is sometimes called alpha testing [97]. The alpha testing process is performed until the system developer and the client agree that the system meets their requirements. Then, a testing process called beta testing is used after system is ready to be used as a software product. Beta testing involves delivering a system to a number of interested customers who will be report issues to the system developers.

2.4 Testing in Software Maintenance

There are several process models proposed for software maintenance including those from IEEE and ISO. The IEEE Standard 1219-1998 [3] has proposed a software maintenance process that has seven phases as shown in Figure 2.5. The phases are:

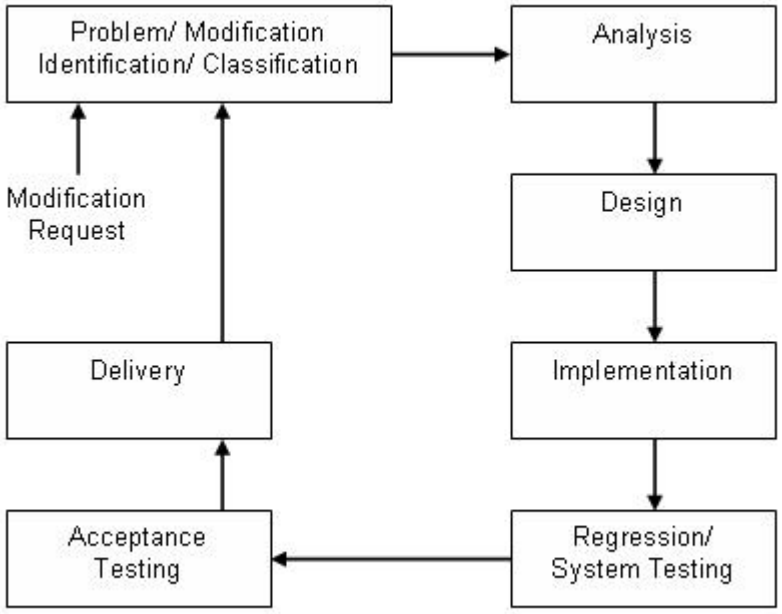

Figure 2.5: IEEE Standard 1219-1998 Software Maintenance Process [3]

- Problem/Modification Identification and Classification

  This is the beginning phase of software maintenance process which starts with the Modification Request (MR). MR can be issued by a user, a customer, a programmer, or a manager. Then, the MR will be assigned with a suitable maintenance type, priority and unique identifier. The phase also has an activities which determine whether to accept or reject the MR. The

maintenance process will be proceed to the next phase or might be stopped here.

- Analysis Phase

  In this phase, a preliminary plan for design, implementation, testing, and delivery will be built according to the conducted analysis. Analysis is performed at two levels: feasibility analysis and detailed analysis. Feasibility analysis identifies alternative solutions and assesses the impact and costs. Detailed analysis defines the requirements for the change, arranges a test strategy, and prepares an implementation plan.

- Design Phase

  This phase designs a modification to the system. The design phase needs to use all current system and project documentation, existing software and databases, and the output of the analysis phase. Some of activities in design phase include identifying the affected parts of software, modify the documentation of software, create a test cases for the new parts of the software, and identifying regression tests for a new version of the software.

- Implementation Phase

  This Phase includes the activities of coding and unit testing, integration and testing of the modified code, risk analysis, and review. The Phase also includes a test-readiness review to asses preparedness for system and regression testing.

- Regression/ System Testing

  This is the phase in which the entire system is tested to ensure compliance to

the original requirements plus the modifications. In addition to functional and interface testing, this phase includes regression testing to validate that no new faults have been added. Regression testing is one of the important activity in the software maintenance because it can reflect the overall costs and resources used in software maintenance. The detailed explanation of regression testing will be discussed in Chapter 3. Finally, this phase is responsible for verifying preparedness for acceptance testing.

- Acceptance Testing

  This phase has a same process as acceptance testing in software development. Its concerned with the fully integrated system and involves users, customers, or a third party designated by the customer. Acceptance testing comprises of functional tests, interoperability tests, and regression tests.

- Delivery Phase

  This is the phase in which the new version of modified systems is released for installation and operation. It includes the activity of notifying the user community, performing installation and training, and preparing and archival version for backup.

The problems of software maintenance can be classified into the types corrective, adaptive, perfective and preventive maintenance. Although software systems have emerged in new environments, the problems still occur around four types of software maintenance that need to be tackled. This means, research activities in software maintenance and evolution still need to be explored in order to tackle the modifications of software systems in new environments.

2.5 Summary

This chapter has discussed software testing in general. The explanation is around classification of software testing, software testing processes and testing in the software maintenance process. These are the important basic knowledge of software testing in order to explore in more detail on regression testing that focussed in this thesis.

Chapter 3

Regression Testing

3.1 Introduction

According to the IEEE Standard 1219-1998 [3], regression testing can be involved in different levels such as unit, integration or system level testing. Li [71] also described regression testing as one kind of testing that is applied at all these three levels. These three levels of testing are similar to the process of testing in development although they have to be focussed on modifications that have occurred in the program. Most existing regression testing techniques concentrate on unit testing. Some of the techniques focused on all levels of testing [89; 102].

This chapter discusses regression testing and specifically regression test selection. The chapter is organised as follows. The second section presents an evaluation framework for regression test selection techniques. The third section presents regression testing strategies. Then, categories of regression testing techniques are discussed in the forth section. The most significant topic in this chapter

is about regression test selection techniques presented in the fifth section. Then, the sixth section discusses a regression testing in different environments.

3.2 An Evaluation Framework for Regression Test Selection Techniques

Rothermel and Harrold [88] proposed a framework for evaluating regression test selection techniques. This framework is used to evaluate the proposed model in the later chapters of the thesis. The framework is based on four categories. They are:

1. Inclusiveness

   Inclusiveness measures the capabilities of techniques to select test cases that will cause the *modified* program to give a different output than the *certified* program. A regression test selection technique is *safe* if it selects all test cases that can give different output.

2. Precision

   Precision measures the ability of techniques to avoid select test cases that cannot give different output between the *certified* and the *modified* programs. A regression test selection technique is *precise* if the technique is capable of omitting test cases that cannot give different output.

3. Efficiency

   Efficiency measures the computational cost, thus the practicality of a regression test selection technique.

4. Generality

   The *generality* of a regression test selection technique is its ability to be used in a wide and practical range of situations.

3.3 Regression Testing Strategies

An important issue in regression testing is how to reuse the existing test suite for the *modified* program. There are two main regression testing strategies; *retest all*, and *selective retest* [89]. A *retest all* approach reruns all the existing test suite on the *modified* program. In theory, *retest all* approach is *safe* because it can exercise all modification parts in the *modified* program. However, it is not practical to use for large software systems because of the time and resources needed.

Selective retest techniques, in contrast, attempt to reduce the time required to retest a *modified* program by selecting a subset of the existing test suite and retesting only the relevant part of the *modified* program. Rothermel and Harrold [89] have identified two issues in the selective retest techniques: (1) the issue of how to select test cases from the existing test suite and (2) the issue of identifying where additional test cases may be required. Both issues have been tackled in the proposed model presented in this thesis.

3.4 Categories of Regression Testing Techniques

Rothermel et al. [92] consider three techniques for reducing the cost of regression testing. They are *regression test selection*, *test suite minimization* and *test case*

prioritization techniques.

3.4.1 Regression Test Selection

Many papers concentrate on regression test selection techniques [10; 21; 22; 28; 43; 44; 51; 53; 54; 55; 89; 90; 102; 103; 110]. Those techniques attempt to reduce the cost of regression testing by selecting appropriate test cases using information from the *certified* program, the *modified* program and the existing test suite. A detailed explanation about this category will be given in the next section.

3.4.2 Test Suite Minimization

Test suite minimization techniques decrease cost by minimizing a test suite that still maintains the same coverage of the initial test suite with respect to a particular test coverage metric. Harrold et al. [49] propose a minimization technique that helps to manage a test suite by determining redundant and obsolete test cases. The technique introduced a mechanism that selects a set of test cases from the test suite, but still provides the desired testing coverage of the program. The technique requires an association between the test cases and the testing requirements of the program, but it is independent of the test selection criteria and can be applied if this association can be made. The minimization technique can also accommodate test suites that use more than one test selection criteria. The technique can be performed on the entire test suite or on a test suite consisting of those test cases that test the changed or affected parts of a program. This technique was incorporated into a data flow testing system called Combat [52]. Hsu and Orso [60] have developed a general framework and tool for supporting

test-suite minimization called MINTS. Their evaluation shows that MINTS can be used to instantiate a number of different test-suite minimization problems and efficiently find an optimal solution for such problems using different solvers [60].

3.4.3 Test Case Prioritization

Many papers concentrate on test case prioritization [20; 31; 32; 63; 67; 68; 72; 75; 76; 83; 84; 91; 92; 98; 109]. Test case prioritization technique provides another method for assisting with regression testing. The prioritization technique let testers order their test cases, so that those test cases with the highest priority are executed earlier than those with lower priority according to some criterion [92]. Elbaum et al. [32] consider 14 test case prioritization techniques classified into three groups. The groups are based on control, statements and function level of a program.

3.5 Regression Test Selection Techniques

The subject of selective regression testing has received considerable attention from the software testing and software maintenance research communities. Some of the regression test selection techniques are discussed below. These regression test selection techniques can be divided into few categories based on elements used in their techniques such as control-flow based [89], textual differencing based [102; 103], code entities based [21] and program slicing based [14; 35; 41; 107].

3.5.1 Control-flow Based

Rothermel and Harrold [89] propose a *safe* and efficient regression test selection technique based on control-flow graphs (CFG). They have proposed two main algorithms; intraprocedural and interprocedural. The intraprocedural algorithm operates on individual procedures. The interprocedural algorithm operates on entire programs or subsystems. In this technique, both the *certified* and the *modified* programs will be transformed into a CFG in order to perform comparison. The comparison algorithm compares each node in both CFGs. If both nodes differ, the algorithm will select test cases from Test Suite (T) that execute the node in CFG of the certified program to test the modified program.

These two algorithms are implemented in two different tools. They are *DejaVu1* for intraprocedural algorithm and *DejaVu2* for interprocedural algorithm. Both tools have been developed to analyse C programs. By using both algorithms, this technique is suitable for a level of regression testing including unit, integration and system level.

Rothermel and Harrold claim that their technique can decrease the time required to carry out regression testing for the *modified* program, even when considering the cost of performing the analysis to select the test cases. Their interprocedural test selection algorithm can give huge savings than intraprocedural test selection algorithm in term of reducing the number of test cases. The technique can give significant savings when applied to large or complex programs. This result is based on their experiment of the application of their technique

to the "Siemens programs" by Hutchins [61]. The result show that DejaVu1 which perform intraprocedural algorithm always selected 100% of test cases for the modified procedures. This means there is no significant reduction in the size of test suite for the modified procedures. In contrast to this, *DejaVu2* in average selects about 55.6% test cases for the modified program. This means *DejaVu2* can give saving about 44.4% of test cases size. This technique is considered as a *safe* regression test selection technique but not *precise* [89; 90].

3.5.2 Textual Differencing Based

Vokolos and Frankl [102] have developed a tool called *Pythia* that is used to reduce the cost of regression testing. The Unix-based tool implements an analysis technique that is called textual differencing because it works by comparing the source files from the *certified* and *modified* programs. The *Pythia* tool can be used to analyse software systems written in the C programming language. Vokolos and Frankl claimed that a novel characteristic of *Pythia* is that it has been implemented by using standard Unix tools. The characteristics of the *Pythia* tool are:

(i) It selects a *safe* regression test suite.

(ii) It supplys both intraprocedural and interprocedural analysis. So, it can be used for single *C* functions or software systems.

(iii) It has been implemented using standard Unix tools.

(iv) The comparison between the *certified* and the *modified* programs uses the Unix tool called *diff*. No abstract representation of the program is needed

in the comparison.

(v) Intrumentation, for determining the execution trace of the *certified* program, is done directly by the *C* compiler, during module compilation.

(vi) In principle, it can be easily extended to support other popular programming languages, such as C++.

The *Pythia* tool has been integrated into a shell script to include *cc*, the C language compiler, *pretty*, a beautifier for C programs, and *diff*, the general purpose file comparison program. Pythia consists of a few stand-alone programs: *kform*, *instr*, *xqt*, and *txt*. The functionality of these programs and a description on how *Pythia* works is as follows:

(i) The sources file for the *certified* program is converted using the program *kform*– into a canonical form. *Kform* is a script that uses the program *pretty*, the C program beautifier.

(ii) The canonical files are instrumented and compiled using the program *instr*. Intrumentation is used to maintain a basic block execution trace for the *certified* program. *Instr* is a script that uses *cc*, the C compiler.

(iii) The program being tested is executed via the program *xqt*, which maintains a history of test cases along with the basic blocks executed by each test case.

(iv) The modified program are also converted into canonical files with the program *kform*.

(v) The program *txt* compares the *certified* program with the *modified* program canonical files, by using *diff*, and analyses the differences, as reported by *diff*, to determine the set of all test cases that have exercised by the modified statements.

Vokolos and Frankl [102] have used the framework for evaluating selective regression testing techniques developed by Rothermel and Harrold [88]. They have claimed that textual differencing is a *safe* selective regression testing technique in terms of inclusiveness. For precision, textual differencing is not 100% *precise* due to the fact that they do not perform semantic analysis. In term of efficiency, the computational cost of textual differencing will be reasonable. In term of generality, textual differencing involves all forms of code modifications like insertions, deletions, and changes of statements. It can works on both in intraprocedural and interprocedural aspects of a program. They also claimed that their technique can easily be extended to programs written in languages that have a mechanism to perform basic block instrumentation and to transform the source code into canonical form.

Vokolos and Frankl [103] claimed that the *Pythia* tool can quickly analyse software systems written in C programs and be effective in reducing the set of regression test cases. The claim is based on the results from a case study involving a software system of approximately 11,000 lines of source code written for the European Space Agency. The system called *ORACOLO2* is written in C and was developed within the Microsoft Visual C++ 1.5 environment. There were 33 different faults discovered and recorded. Each fault was corrected and a new

version of the program was created for each fault. The results of their case study shows that Pythia reduced the size of the regression test suite by at least 90% on average in almost 40% of the program versions (13/33). A reduction of at least 80% was reported in almost 50% of the program versions (16/33). This shows that the textual differencing based technique, Pythia, can give significant reduction in regression test suite size. Pythia is considered as a *safe* regression test selection technique but not *precise* [103].

3.5.3 Code Entities Based

Chen et al. [21] have proposed a regression test selection technique based on identifying modified code entities such as functions, variables, types, and macros. Test cases that have traversed modified code entities will be counted in the test suite for the *modified* program. The technique has been implemented in a tool called *TestTube* that combines static and dynamic analysis to perform selective retesting of programs or systems written in the C programming language. The tool has been developed with a combination of existing analysis tools. The collection of tools can be divided into three categories, including instrumentation tools, program database tools, and test selection tools. In the instrumentation tools, *app* (the Annotation Preprocessor C) instruments the source code automatically. The C Information Abstractor (CIA) is used to build a C program database in the program database tools category. The technique is considered as a *safe* regression test selection technique but less *precise* [88].

3.5.4 Slicing Based Techniques

There are a number of regression test selection techniques based on program slicing techniques. The concept of program slicing will be explained in detail at Chapter 4. Binkley [14] conducted a survey about the application of program slicing to regression testing. He divided into three groups of program slicing that are used in regression testing. The first group uses dynamic slicing, the second group presents program slicing using program dependent graphs (PDG), and the third group is based on Weiser's data-flow definition of slicing [104].

Agrawal et al. [7] have proposed three algorithms to be used in their technique called an incremental regression testing. The algorithms are an execution slice, a dynamic slice, and a relevant slice. The execution slice of the program with respect to a test case is referred to as the set of statements executed under that test case. The dynamic program slice with respect to the output variables gives us the statements that are not only executed but also have an effect on the program output under that test case. The relevant slice with respect to the program output for a test case is referred to the set of statements that, if modified, may alter the program output for the given test case.

Agrawal et al. [7] have pointed out that the amount of regression testing effort saved using their technique obviously depends on the nature of test cases as well as the locations of the modifications made. If the number of test cases are large and each of them exercise small parts of the program's functionality then using these techniques should offer huge savings. The modification parts of the

program may also have a major effect on the amount of savings implied by using these techniques. The incremental regression testing technique is considered as a *precise* regression test selection technique but less *safe* [10].

Gupta et al. [45] have developed a data flow based regression testing technique that uses slicing algorithms to explicitly determine the affected *definition-use* associations made by a program change. The technique uses two slicing algorithms to detect directly and indirectly affected *def-use* associations. The first algorithm works backward from the changed statement to its definitions. The second algorithm is a forward walk from the same point as the first algorithm. The forward algorithm detects uses, and subsequent *definitions* and *uses*, that are affected by a definition that is changed at that point.

Gupta et al. [45] claim that the slicing algorithms are efficient because they detect the *def-use* associations without considering either the data flow history or the complete recomputation of data flow for the *certified* program. They also claim that their technique could easily be modified from *all-uses* criterion to other data flow testing criteria. The technique can also be extended to interprocedural regression testing using interprocedural slicing. The technique is considered as a *safe* regression test selection technique but less *precise* [88].

Gallagher et al. [40] have proposed a novel approach for regression test selection based on *exclusion*. They claim that an exclusion-based technique is likely to be more effective that an inclusion-based technique in two ways. First, it will more confidently identify all non-modification revealing tests in terms of safety.

Second, in terms of the impact of the approach, by reducing the size of regression tests by excluding tests that are not related to modification. Gallagher et al. proposed four steps in his exclusion technique as follows:

1. Decompose and *Reduce* System Version n. The decomposition slices are constructed for the considered system and reduced by equivalent slices.

2. Match Tests with Code. The decomposition slices are match to the relevant test cases using Vokolos and Frankl technique [102].

3. Decompose and *Reduce* System Version $n + 1$. The process is same as in step 1. Then, obtain the tests for decomposition slice clusters that remain unchanged.

4. Use tests that remain after removing those obtained in step 3. Any tests for unchanged code are not needed.

3.6 Regression Testing in Different Environments

There are implementations of regression testing techniques in the literature. They can be divided into four groups: structured based programs, object-oriented based programs, web based applications and component-based systems.

3.6.1 Structured Based Programs

Structured based program are often composed of program flow structures such as sequence, selection and iteration compare to object-oriented program that are based on objects which have their attributes and methods. There are a number

of techniques as well as tools that are proposed for regression testing for structured based programs, especially the C programming language. Examples are the Rothermel and Harrold technique with their tools *DejaVu1* and *DejaVu2* [89], *TestTube* tool by Chen et al. [21], and *Pythia* tool by Vokolos and Frankl [102]. The explanation of these techniques and tools have already been described in the previous section.

3.6.2 Object-oriented Based Programs

Orso et al. [79] have introduced a regression test selection technique for Java programs. The technique can handles the object-oriented features of the language, is *safe* and *precise*, and applicable to large systems. The technique consists of two parts: partitioning and selection. The partitioning part is executed first in order to build a high level graph representation of *certified* and *modified* programs and performs an analysis of the graphs. The goal of the analysis is to identify the parts of the *certified* and the *modified* programs that have changed based on information on changed classes and interfaces. Then, the selection part of the technique builds a more detailed graph representation of the identified parts of the *certified* and the *modified* programs, analyses the graph to identify differences between the programs, and selects a set of test cases in the test suite that traverse the changes. This technique is implemented in a tool called *DEJAVOO*. Orso et al. claim the results of the empirical study of their tool is encouraging in terms of efficiency and effectiveness. The technique reduces the time for regression testing as high as 62.5% for a largest system. The cost-effectiveness improves with the size of the program under test.

Wu et al. [106] have proposed a regression testing technique based on the analysis of the dependence relationship among functions in a system. They have defined that the object-oriented features, such as inheritance, dynamic binding, polymorphism and message passing are related to the function calls which are associated with certain objects. The technique performs in two phase analysis. The first phase is to analyse the affected variables, functions, function dependence relationships at the statement level after the modification. The technique is *safe* because it considers all possible effects of the modification on the system. This static phase is considerably more efficient. In the second phase, the technique dynamically select test cases that are needed to be retested by using the function calling graph (FCG) of each test case in order to precisely process object-oriented features and thus enhance the precision of the technique. The FCG can be constructed based on the record of the calling sequence of functions. So, the required overhead is proportional to the number of function calls.

Harrold et al. [50] have introduced a *safe* regression test selection technique for Java. The technique can efficiently handle the features of object-oriented language specifically the Java language, such as polymorphism, dynamic binding, and exception handling. The technique is an adaptation of Rothermel and Harrold technique [89], which is based on a control flow representation of the *certified* and *modified* programs to select test cases to be rerun. The technique performs three steps. First, it constructs a graph to represent the control flow and the type of information for the set of classes under analysis. Then, it traverses the graph to identify affected edges. Finally, based on the coverage matrix obtained

through instrumentation, the technique selects the test cases that exercise the affected edges identified from the test suite for the *certified* program.

Unlike the Rothermel and Harrold technique [89], which is uses the CFG, the technique by Harrold et al. [50] introduces the Java Interclass Graph (JIG) as a representation of the program. A JIG accommodates the Java features and ca be used by the graph-traversal algorithm to identify dangerous entities. Dangerous entity is an edge that affected by a change by comparing the *certified* and the *modified* programs. Empirical studies indicate that the technique can be effective in reducing the size of the test suite [50].

3.6.3 Web Based Applications

Tarhini et al. [99] have proposed a safe regression testing selection technique for web applications based on an Event Dependency Graphs (EDG). The EDG is used to model the *certified* and the *modified* web applications. Then both EDG's are compared in order to select the affected nodes and the potentially affected nodes. The affected nodes are used to select test suite for the *certified* web application. Empirical results show that the technique reduced the test set size [99]. About 44-90% of test cases were eliminated. The selected test cases still cover the modified and potentially modified components.

Lin et al. [73] have introduced a code transformation approach to regression test selection. The transformed code forms a local Java program which simulates the functionality and behavior of the Web service applications in an end-to-end

manner. Safe regression test selection techniques can then be applied to the transformed code and safely reduce the test cases for the Web service applications. This approach is implemented on Web service applications written in Java and deployed in the Axis server only.

Ruth et al. [93; 94] have proposed a gray-box approach that support *safe* regression test selection technique for verification of Web service system in an end-to-end manner. A gray-box approach is a technique that does not involve code-based knowledge directly, in contrast to white box approach. Their approach is based on the *safe* regression test selection technique by Rothermel and Harrold [89] which is uses a CFG as a representation of the *certified* and *modified* programs. Each node represents a code entity and each edge represents the control flow from one code entity to another. The entities can be statements, methods, classes, or components [94]. Then, the technique identifies affected edges by comparing the CFGs of *certified* and *modified* programs. Finally, based on the set of affected edges, the technique selects test cases for T' from test suite T that need to be rerun.

3.6.4 Components Based System

Gao et al. [42] have proposed a systematic retest method for software components based on a component retest model. This method has been implemented in a component test tool called *COMPTest*. The *COMPTest* tool can automatically identify component-based API changes and impacts, as well as reusable test cases in a component test suite. Gao et al. claimed that the tool has two major

advantages:

(i) Automatic identification and analysis of API-oriented component changes and impacts based on given API-based component test models and other meta-data, such as function and dependency information in a component.

(ii) Automatic black-box test selection for reuse and test suit refreshment for a component.

3.7 Summary

This chapter mainly focuses on regression test selection techniques. The chapter has started with definition of regression testing. Then, the chapter discusses the evaluation framework for regression test selection techniques, regression testing strategies and categories. Finally, the chapter explains the applications of regression testing in the different environments. These are basic knowledge that are important in order to understand regression test selection techniques that are necessary for the research in this thesis.

These regression test selection techniques attempt to reduce the cost of regression testing by selecting appropriate test cases using information from the *certified* program, *modified* program and test suite. The techniques are classified as inclusion techniques which select test cases from test suite that are needed in regression testing. There is no existing techniques that are based on exclusion technique. The idea of the regression test selection by exclusion is proposed by Gallagher et al. [40]. Exclusion technique omits test cases from test suite that are not needed in regression testing.

Chapter 4

Program Slicing

4.1 Introduction

Program slicing was first introduced by Weiser in 1981 [104]. Since then, program slicing has grown and become an important research field in software engineering. This fact was endorsed by Binkley and Gallagher [15], who stated that the number of citations for the paper by Weiser on program slicing increased significantly year by year. Recently, there are a number of papers that have done a survey on program slicing techniques and it applications [29; 38; 74; 96]. Since Weiser's first program slicing technique, many program slicing techniques have been introduced such as dynamic slicing [6; 69], forward slicing [12], decomposition slicing [41], interprocedural slicing [59], conditioned slicing [18], stop-list slicing [39], amorphous slicing [16], hybrid program slicing [86] and abstract slicing [58; 108].

Program slicing is a decomposition technique that produces a new sub-program relevant to a particular computation. The new sub-program is called a *slice*, and

is an executable program that is produced from the original program with respect to the specified *slicing criterion*. *Slicing criterion* is a set of conditions used in the slicing computation to produce a slice. A basic *slicing criterion* uses two main parameters. They are a variable or a set of variables and the location of interest.

This chapter is organised in six sections. The next section discusses the representation of programs or systems. This is followed by a discussion of program slicing techniques in the third section. The fourth section discusses the decomposition slicing technique that is used in the model proposed in this thesis. The fifth section is about the applications of program slicing.

4.2 Representations of Program

Tip [100] states that Weiser's approach uses data flow and control flow dependences in order to compute a slice. There are other different representations used in different types of slicing such as control flow graphs, program dependence graph, and system dependence graph. A brief explanation of these representations is given below.

4.2.1 Control Flow Graph

A Control Flow Graph (CFG) is a representation of the program with the combination of nodes and edges from the start node to the end node. A CFG represents control dependencies of the program. Nodes in the graph are the program statements, while edges represent a flow of control from one to another. In Chapter 2,

the control flow graph shown in Figure 2.3 (page 11) represents the program in Figure 2.2 (page 11).

4.2.2 Program Dependence Graph

A Program Dependence Graph (PDG) is an intermediate representation of a program using a combination of *data dependences* and *control dependences* of the program [34; 59; 80]. *Data dependences* are used to represent data flow relations of the program. *Control dependences* represent control flow relationships of the program. *Control dependences* are derived from the CFG. For instance, in Figure 4.1, statement 7 is dependent on statement 3 because statement 7 has the use of the variable *sum* that depends on its definition at statement 3. The relation of both statements is called *data dependence*. Statements 5 and 7 show the relationship between statement and predicate. Statement 7 is dependent on statement 5 as a predicate. This dependence is called *control dependence*. An example of PDG is shown in Figure 4.2 [59]. The bold arrowed lines represent control dependence edges and the other arrowed lines represent data dependence edges.

4.2.3 System Dependent Graph

Horwitz et al [59] have introduced the concept of System Dependence Graph (SDG). SDG is an extension of the PDG. It includes the PDG, which represents the main program of the system; procedure dependence graphs, which represent the procedures of the system; and some additional edges. There are two types of additional edges. These are edges that represent direct dependences between a

```
(1)  read (n);
(2)  i := 1;
(3)  sum := 0;
(4)  product := 1;
(5)  while i <= n
(6)  {
(7)    sum := sum + 1;
(8)    product := product * i;
(9)    i ++;
(10) }
(11) write (sum);
(12) write (product);
```

Figure 4.1: The Program to be Sliced [100]

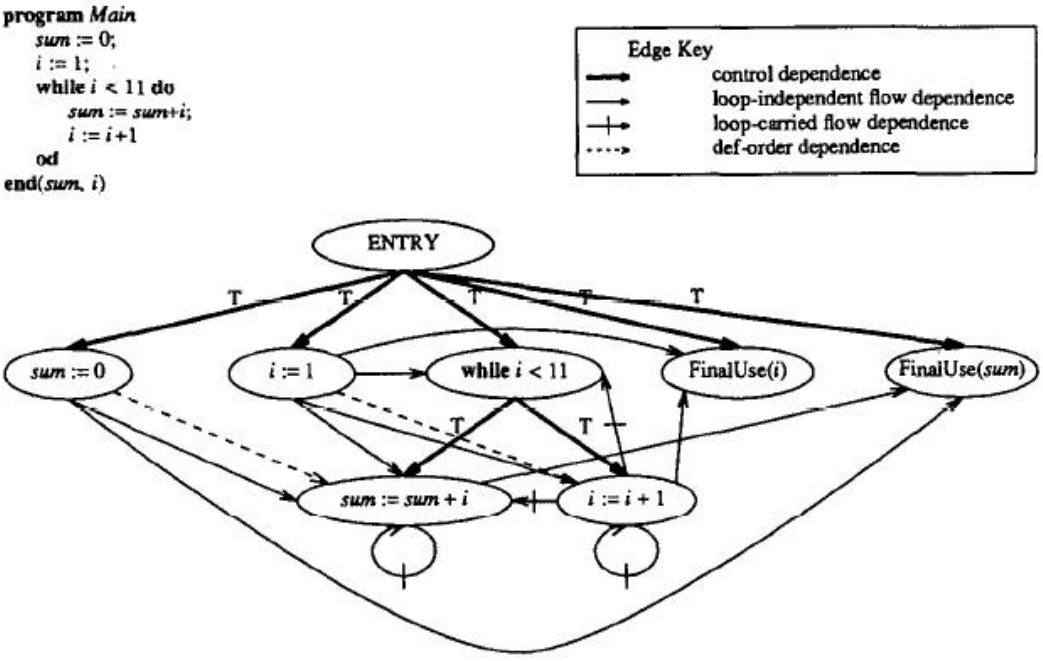

Figure 4.2: Program Dependence Graph [59].

call site and the called procedure, and edges that represent transitive dependences due to calls. An example of the SDG for program in Figure 4.3 is shown in Figure 4.4 [59]. Transitive interprocedural flow dependences are represented by using heavy bold arcs. The call edges, *parameter-in* edges, and *parameter-out* edges which connect program and procedure dependence graphs together are represented by using dashed arrows.

```
program Main          procedure A(x, y)   procedure Add(a, b)  procedure Increment(z)
 sum := 0;             call Add(x, y);     a := a + b           call Add(z, 1)
 i := 1;               call Increment(y)   return               return
 while (i<11) do      return
  call A(sum, i)
 od
end
```

Figure 4.3: The Program for SDG [59]

4.3 Program Slicing Techniques

The following is a discussion of some program slicing techniques. This includes static and dynamic slicing, backward and forward slicing, conditioned slicing and stop-list slicing.

4.3.1 Static and Dynamic Slicing

The first program slicing technique by Weiser was based on static program analysis [104]. Weiser's program slices are called an *executable static slice* [15]. *Executable* because the slices are an executable program. *Static* because the computation of slices is performed without considering the input of the program. A

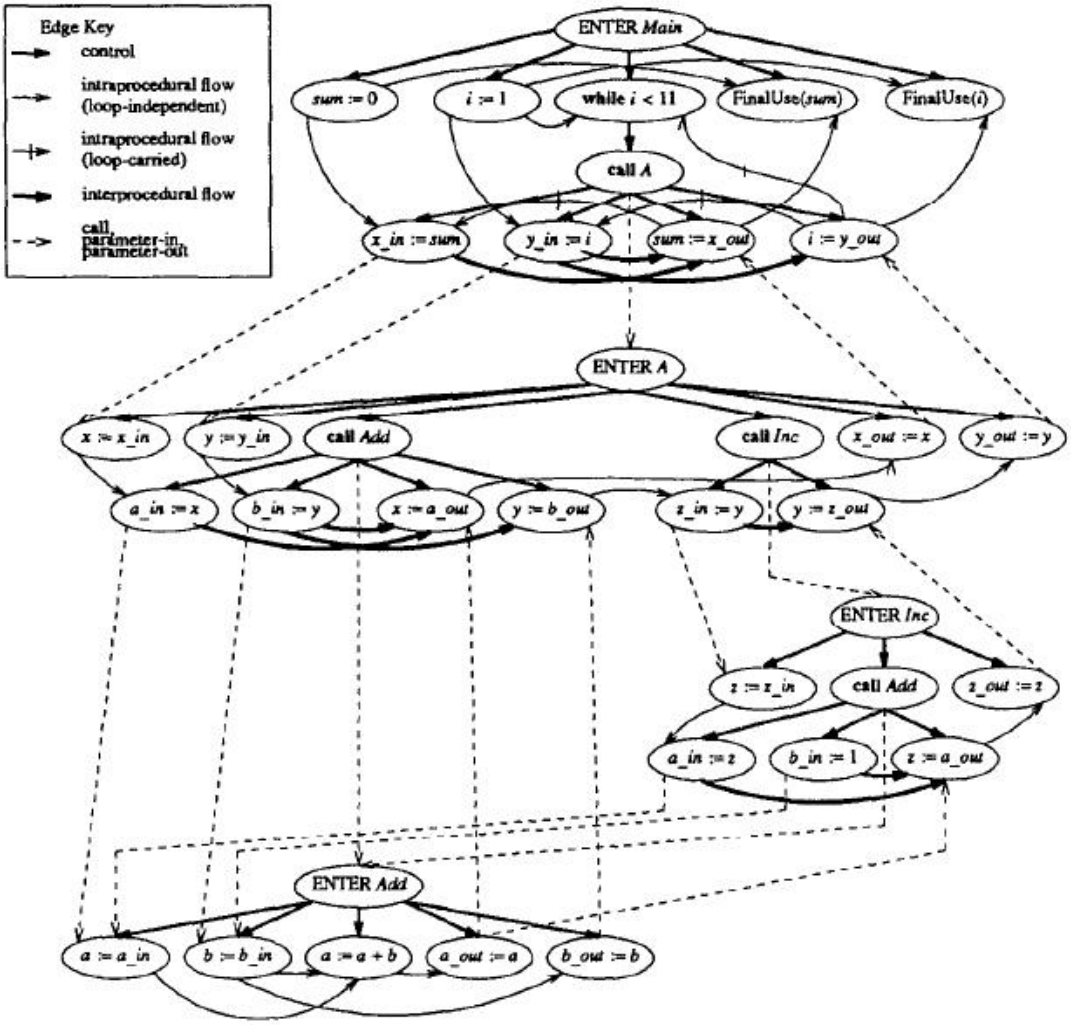

Figure 4.4: System Dependence Graph [59]

basic static slice is shown in Figure 4.5 for the program in Figure 4.1 [100]. Figure 4.1 shows a program which computes the value of variable *sum* and *product* if the input n is a positive number. Figure 4.5 shows the slice of the original program with respect to the slicing criterion (*product*, 12). All statements that are involved in the computation of the variable *product* at line 12 are included in the slice. In other words, all statements that are involved in the computation of the variable *sum* have been excluded from the slice. It shows that program slicing is a useful technique to decrease the size of the program and ease the process of program manipulation at the later stage in any domains of interest.

```
(1)  read (n);
(2)  i := 1;
(3)
(4)  product := 1;
(5)  while i <= n
(6)  {
(7)
(8)     product := product * i;
(9)     i ++;
(10) }
(11)
(12) write (product);
```

Figure 4.5: The Slice of the Program w.r.t Criterion (*product*, 12) [100]

Korel and Laski [69] have proposed dynamic slicing as a counterpart of Weiser's static slicing technique. Their technique has considered the input values in the computation of slice. They introduced the concept of the *trajectory* which is the path that has actually been executed for some input. The concepts of data flow and control flow are used in order to produce *Data-data* (DD) and *Test Control* (TC) relations based on the *trajectory*. The *DD relation* is equivalent to the con-

cept of *definition-use* (*du*) and the *TC relation* is based on control dependence. A dynamic slice can be computed by using the DD and TC relations. The main element in their technique is that they compute a slice based on a program execution (*trajectory*) not a CFG.

Agrawal and Horgan [6] have also discussed dynamic slicing. They have introduced the concept of Dynamic Dependence Graph (DDG) that is based on the PDG. The only difference between them is that the DDG creates a separate node for each occurrence of a statement in the execution history. In other words, the number of nodes in the DDG is equal to the number of statements in the execution history including repeated statements. Figure 4.7 shows a DDG of the program in Figure 4.6 for the test case (N=3, X = -4, 3, -2). Nodes in bold are the dynamic slice for the test case with respect to the variable Z at the end of the execution.

```
S1      read (N);
S2      I := 1;
S3      while (I <= N)
        {
S4         read (X)
S5         if (X < 0)
S6            Y := f1(X);
           else
S7            Y := f2(X);
S8         Z := f3(Y);
S9         write(Z);
S10        I := I + 1;
        }
```

Figure 4.6: The Program for DDG [6]

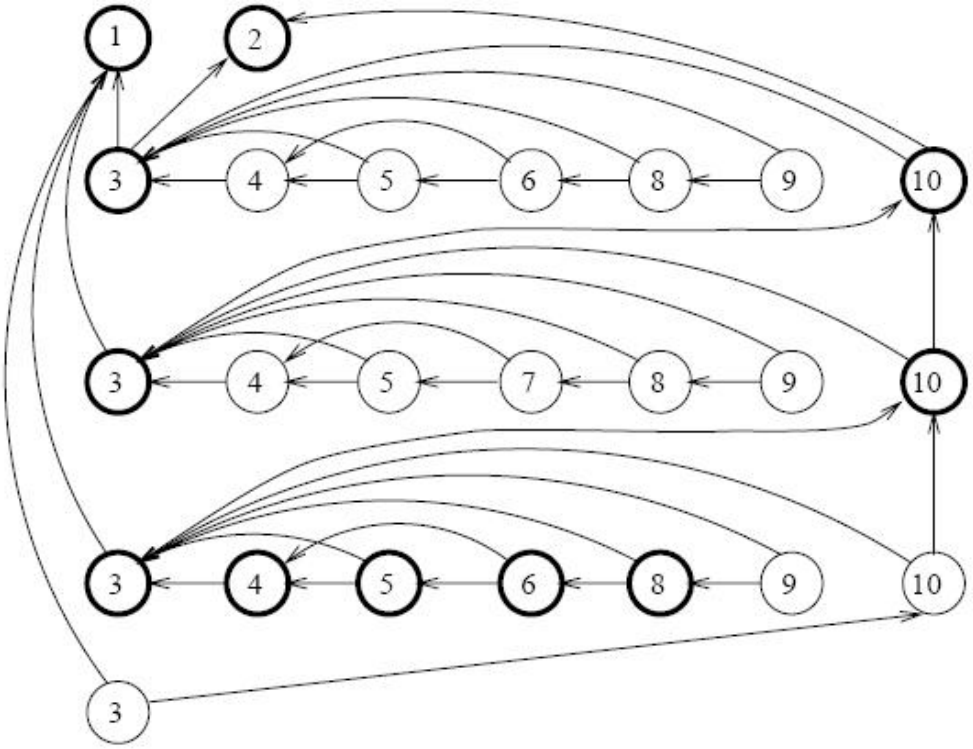

Figure 4.7: Dynamic Dependence Graph [6]

4.3.2 Backward and Forward Slicing

Weiser's program slicing technique is also known as a *backward slicing*. *Backward* because the way edges are traversed using a dependent graph. Weiser's backward slicing computes slices using the data flow analysis that begins by tracing backward the possible statements that have influences on the variable of interest. For example, the slice for the program in Figure 4.1 with respect to the variable *sum* at line 11 is statements 1, 2, 3, 5, 6, 7, 9, 10 and 11. The computation of the slice starts at line 11 which is the *use* of the variable *sum*. From the *use* of this variable *sum*, the slice will be computed backward using the CFG. The last definition (*def*) of the variable *sum* is at line 7. From this line all related

definition-uses are considered in the slice.

Bergeretti and Carre [12] have introduced the notion of forward slicing. Forward slicing includes all statements that depend on the slicing criterion. Forward slice can be obtained from the PDG. Horwitz et al. [59] have computed forward slices for interprocedural program based on the SDG.

4.3.3 Conditioned Slicing

Conditioned program slicing was first introduced by Canfora et al. [18] and later modified as variants [24; 25; 47; 57]. Conditioned program slicing forms a bridge between the static and dynamic analysis. The conditioned slicing criterion is a triple, (p, V, n) where p is some initial conditions of interest and (V,n) are the two elements of the static slicing criterion. For example, the conditioned slice of the program in Figure 4.8 with respect to the criterion, (x >0, {x}, 8) is shown in Figure 4.9 [47].

```
S1    scanf(%d, &x);
S2    y = 2 * x;
S3    if (y > x){
S4        x = x + 1;
S5        y = y * y;}
      else{
S6        x = x * 2;
S7        y = y  x;}
S8    printf(%d, x);
```

Figure 4.8: The Program for Conditioned Slicing [47]

```
S1    scanf(%d, &x);
S2    y = 2 * x;
S3    if (y > x)
S4       x = x + 1;
```

Figure 4.9: The Conditioned Slice [47]

4.3.4 Stop-List Slicing

Early program slicing techniques required two parameters: a variable or a set of variables, and a program location of interest. All statements related to this slicing criterion are included in the program slice. Gallagher et al. [39] have introduced a new technique that has considered a third additional parameter in the slicing criterion. The third parameter is called *stop-list* and is a set of variables that are not of interest. The computation of a stop-list slice will exclude all statements that are related to these excluded variables by using the data-flow dependence analysis. In theory, this technique has the potential to reduce the size of slice compared to the traditional slicing techniques. The evaluation of this technique by Gallagher et al. [39] shows that the results are encouraging giving a large reduction in the slice size.

4.4 Decomposition Slicing

Gallagher and Lyle [41] have introduced the term decomposition slicing. The technique uses slicing to decompose a program directly into two parts, *decomposition slice* and *complement*. The *decomposition slice* is built for one variable and is the union of all slices taken at line numbers of the uses of the given variable. The calculation of these slices can use any independent slicing techniques. There-

fore, the quality of the decomposition slice is dependent on the quality of the slice itself. The *complement* is the sub-program that remains after the decomposition slice is removed from the original program.

A program slice is dependent on a variable and a location of interest. A decomposition slice is only dependent on variables and not the location of interest. For instances, in Figure 4.10 [41], the backward slice with respect to the slicing criterion (t, 4) is {S1, S2, S3, S4}, while the backward slice with respect to the slicing criterion (t, 6) is {S1, S2, S5, S6}. The slice with respect to the variable t at the last statement (S6) is not capable of identifying all the computations involving the variable t. However, the decomposition slice includes all relevant computations involving a given variable without statement numbers. It is produced from the union of both backward slices which includes all statements in the program.

```
S1      input a
S2      input b
S3      t = a + b
S4      print t
S5      t = a - b
S6      print t
```

Figure 4.10: The Program for Decomposition Slice [41]

Figure 4.11 shows a program for calculating the number of lines, words and characters in a text file. There are five decomposition slices available with respect to each variable (*c*, *nl*, *nw*, *nc*, *inword*) in the program. For instance, the decomposition slice for the variable *nw* is shown in Figure 4.12. All relevant statements of the variable *nw* are included in that decomposition slice. All state-

ments that are irrelevant to the variable *nw* are included in the *complement* of the decomposition slice on variable *nw*. This *complement* is shown in Figure 4.13.

```
1       #define YES 1
2       #define NO 0
3       main()
4       {
5           int c, nl, nw, nc, inword;
6           inword = NO;
7           nl = 0;
8           nw = 0;
9           nc = 0;
10          c = getchar();
11          while (c != EOF){
12               nc = nc + 1;
13               if (c == '\n')
14                   nl = nl + 1;
15               if (c == ' ' || c == '\n' || c == '\t')
16                   inword = NO;
17               else if (inword == NO){
18                   inword = YES;
19                   nw = nw + 1;
20               }
21               c = getchar();
22           }
23           printf("%d \n", nl);
24           printf("%d \n", nw);
25           printf("%d \n", nc);
26       }
```

Figure 4.11: The Program to be Sliced [41]

Gallagher and Binkley [37] have discussed decomposition slice equivalence in order to reduce the number of decomposition slices. Their empirical study shows that there can be a significant reduction to the number of decomposition slices by

```
1       #define YES 1
2       #define NO 0
3       main()
4       {
5           int c, nw, inword;
6           inword = NO;
8           nw = 0;
10          c = getchar();
11          while (c != EOF){
15              if (c == ' ' || c == '\n' || c == '\t')
16                  inword = NO;
17              else if (inword == NO){
18                  inword = YES;
19                  nw = nw + 1;
20              }
21              c = getchar();
22           }
24           printf("%d \n", nw);
26       }
```

Figure 4.12: The Decomposition Slice on *nw* (no. of word) [41]

removing equivalence slices. They have used a differencing program to evaluate the decomposition slice equivalence. In their case study, the original program had 95 decomposition slices, removing "empty" slices and combining all equivalent decomposition slices reduced the number to 34, a 62% reduction.

Gallagher et al. [40] have stated that decomposition slicing can be used to divide a program into three parts as shown in Table 4.1. Only the Independent and Dependent statements are of interest for software testing. Any changes in the Independent part will affect only statements in the decomposition slice for variable v. Any changes in the Dependent part will affect statements not only in the decomposition slice for variable v, but also any other relevant decomposition

```
3       main()
4       {
5           int c, nl, nw, nc, inword;
7           nl = 0;
9           nc = 0;
10          c = getchar();
11          while (c != EOF){
12              nc = nc + 1;
13              if (c == '\n')
14                  nl = nl + 1;
21              c = getchar();
22           }
23           printf("%d \n", nl);
25           printf("%d \n", nc);
26       }
```

Figure 4.13: The Complement of Decomposition Slice on nw [41]

slices. The Complement statements cannot be affected by the change. Therefore, decomposition slicing is capable of identifying the unchanged parts of a program and this will be used implicitly in the rest of the proposed model in this thesis.

Table 4.1: Classification of Statements in Decomposition Slice for Variable v

Program Parts	Includes statement which are...
Independent	in the decomposition slice taken with respect to v that are not in any other decomposition slice
Dependent	in the decomposition slice taken with respect to v that are in another decomposition slice
Complement	not independent, i.e. statements in some other decomposition slice (but not v's)

4.5 Applications of Program Slicing

Since Weiser's first program slicing technique, there are a number of the applications of program slicing that have been explored such as debugging, pro-

gram comprehension, software maintenance, software testing, regression testing, program verification, dead code elimination, compiler optimisation, parallelization of sequential programs, showing differences between program, cohesion measurement, clustering equivalent computations and database schema impact [16; 18; 29; 38; 41; 46; 58; 62; 74; 96]. The following are some of the applications of program slicing.

4.5.1 Debugging

The original program slicing technique by Weiser was developed to aid debugging activities [104]. In debugging, the purpose is to identify errors that occur in the program. Program slicing techniques can assist the debugger to detect errors and the affected statements without considering the unrelated statements. Program slicing can minimize the size of the original program to the parts of interest based on the slicing criterion. The application of debugging has also motivated the introduction of dynamic slicing [46]. Dynamic slicing [6; 69] can offer a better assistant in debugging. It can produce a smaller slice compared to static slicing for a specific program input.

4.5.2 Program Comprehension

An early part of the software maintenance phase is program comprehension. Program slicing can be used to assist the program comprehension process. For instance, Canfora et al. [18] have used conditioned slicing in the context of program comprehension and reused an existing software. Conditioned slicing enables the computation of refined code fragments implementing specific program behaviors.

Binkley et al. [16] have used amorphous slicing for program comprehension

4.5.3 Software Maintenance

Software maintenance is always dealing with changes. It determines whether a change at some parts of the program will affect the behavior of the other parts of the program. Program slicing can be used in order for the maintainer to concentrate only on the modified parts of the program. This can minimize the chances of introducing unexpected errors. Gallagher and Lyle [41] have introduced decomposition slicing that was used in a new software maintenance process model.

4.5.4 Software Testing

There are two main structural based testing techniques: control flow testing and data flow testing. Program slicing techniques are based on the manipulation of control flow and data flow graphs. The important part of software testing that applies program slicing techniques is regression testing. Slicing based regression test selection techniques have been discussed in the previous chapter.

4.6 Summary

This chapter has focused on program slicing. The chapter starts with the definition of program slicing as presented in Section 4.1. Section 4.2 discusses the representation types of the programs. Section 4.3 explains the program slicing techniques. The main focus in this chapter is the decomposition slicing technique that is used in the proposed model as presented in Section 4.4. Finally, the

applications of program slicing are explained in Section 4.5. All these sections are important in order to understand program slicing, specifically decomposition slicing technique. This slicing technique is capable of identifying the unchanged parts of a program, and this will be used implicitly in the rest of the proposed model in this thesis.

Chapter 5

Regression Test Selection by Exclusion (ReTSE)

5.1 Introduction

This chapter proposes a novel Regression Test Selection by Exclusion (ReTSE) model using the decomposition slicing technique. The ReTSE model produces an optimised regression test set for a new version of a program that has been modified from a previous version. The model is only designed for a program that compiles and runs properly.

The chapter is organised as follows. The ReTSE model is presented in the second section which has four sub-sections that describe each phase in the model. The sub-sections are Program Analysis (which includes Pretty Print and Slicing), Comparison, Exclusion and Optimisation. The third section illustrates the model using a simple example.

5.2 The Model

A high level view of the ReTSE model is shown in Figure 5.1. There are three inputs of the model. They are the Original Certified Program (OC) which has previously been tested, the Original Modified Program (OM) which is a new version of the program and the existing Test Suite (TS) which includes test cases and its test histories. Test history is a set of statements executed for a particular test case. The outputs of the model are a set of Excluded Tests (ET), a set of Optimised Regression Tests (RTO) and, in some instances, request for new test cases. The ET is a set of test cases that do not need to be used for regression testing and RTO is a set of test cases that have been reduced from a set of Regression Tests (RT).

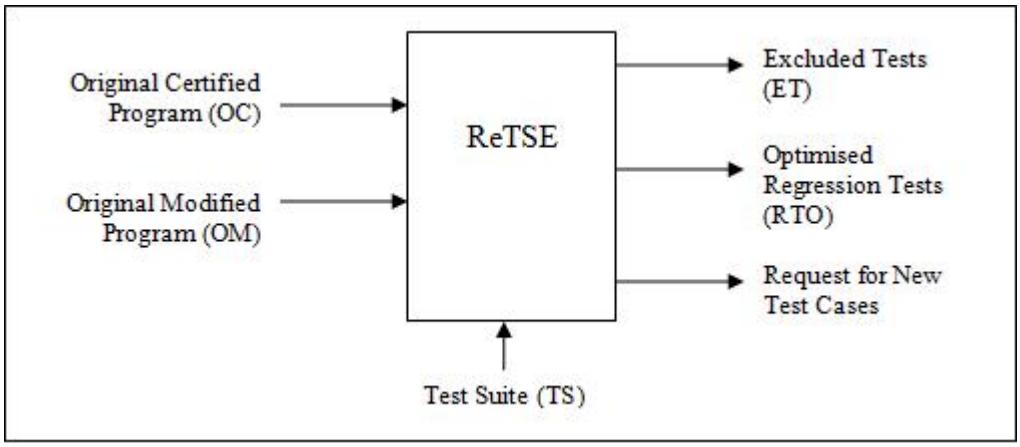

Figure 5.1: The ReTSE Model - High Level

The ReTSE model has four main phases: 1. Program Analysis, 2. Comparison, 3. Exclusion and 4. Optimisation, as shown in Figure 5.2. The purpose of each phase is described together with the expected input and output.

5.2.1 Phase 1: Program Analysis

The Program Analysis Phase manipulates and analyses the program into a form suitable for use in later phases. This phase has two steps as shown in Figure 5.3. They are:

- 1.1 Pretty Print
- 1.2 Slicing

5.2.1.1 Step 1.1: Pretty Print

Input:

- Original Certified Program (OC)
- Original Modified Program (OM)

Output:

- Certified Program (C)
- Modified Program (M)

Generally, Pretty Print is the process of formatting a program into some standard forms. It consists of changes to positioning, spacing, blank lines, commented lines, indentation, white spaces and other similar modifications. Both the Original Certified and Original Modified Programs (OC and OM) will be transformed

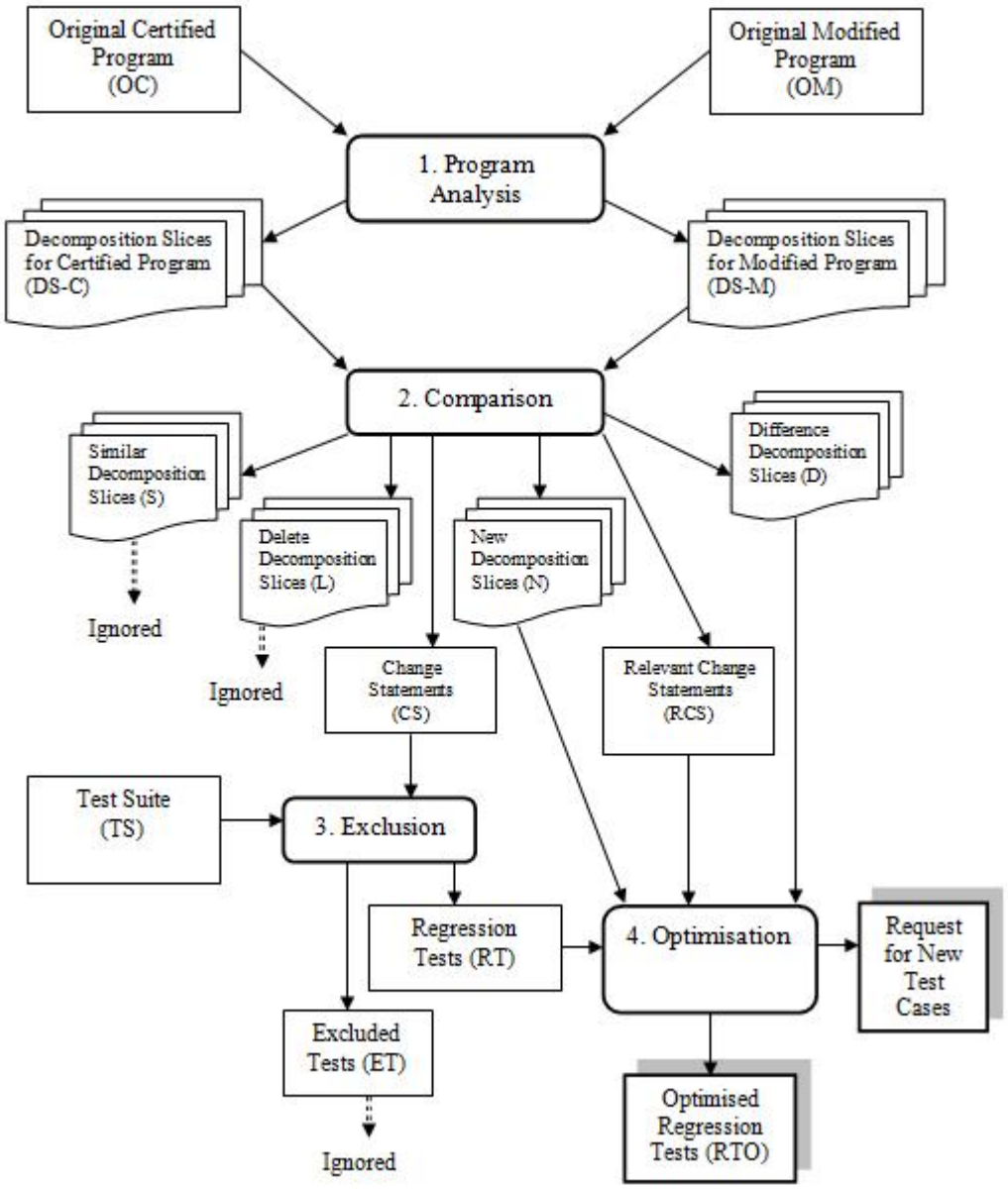

Figure 5.2: The ReTSE Model -Low Level

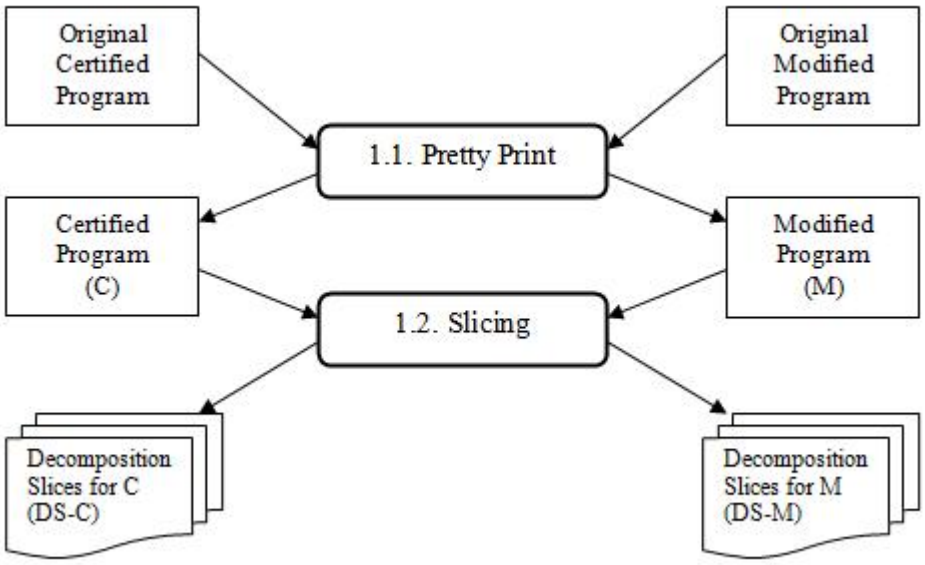

Figure 5.3: The ReTSE Model - Program Analysis

into a standard format called a Certified Program (C) and a Modified Program (M) respectively. This step is included to standardise the layout of the program and to make the Comparison Phase later in the model easier to perform. The new layouts of the programs have the following characteristics:

(i) Only one statement in the line.

(ii) No comment lines.

(iii) No split lines for single statements.

(iv) An indentation is used for:

- the branch of an *if* statement.
- the body of a while and *for* statement.

(v) A curly bracket ({}) is introduced for:

- the branch of an *if* statement with one statement at a time.
- the body of a *while* and *for* statement with one statement at a time.

5.2.1.2 Step 1.2: Slicing

Input:

- Certified Program (C)
- Modified Program (M)

Output:

- Set of Decomposition Slices for C (DS-C)
- Set of Decomposition Slices for M (DS-M)

In Slicing, both the Certified Program (C) and the Modified Program (M) will be sliced using the decomposition slicing technique. The number of decomposition slices of C and M corresponds to the number of variables in the program. If a variable is declared but not used then the decomposition slice will be empty. In this model, backward slicing is used in the decomposition slicing because it is capable of identifying all relevant statements of the given variable. A summary of this Slicing step is shown in the following notations:

- Set of Decomposition Slices for C (DS-C)

  Let program C have v_c variables.

  DS-Cv_i : The decomposition slice for variable v_i in the C.

  DS-C : The set of decomposition slices for all variables v_i in the C.

  So, DS-C = {DS-Cv_i | i = 1, 2,... v_c}

- Set of Decomposition Slices for M (DS-M)

  Let program M have v_m variables.

  DS-Mv_i : The decomposition slice for variable v_i in the M.

  DS-M : The set of decomposition slices for all variables v_i in the M.

  So, DS-M = {DS-Mv_i | i = 1, 2,... v_m}

5.2.2 Phase 2: Comparison

Input:

- Set of Decomposition Slices for C (DS-C)
- Set of Decomposition Slices for M (DS-M)

Output:

- Set of pairs of Similar Decomposition Slices (S)
- Set of pairs of Difference Decomposition Slices (D)
- Set of Delete Decomposition Slices (L)
- Set of New Decomposition Slices (N)
- Set of Change Statements (CS)

- Set of Relevant Change Statements (RCS)

There are two parts in the Comparison Phase. The first part is a comparison between the DS-C and the DS-M. Here, the comparison is at the textual level. The results of this comparison will be:

(i) Set of pairs of Similar Decomposition Slice (S)

(ii) Set of pairs of Difference Decomposition Slice (D)

(iii) Set of Delete Decomposition Slice (L)

(iv) Set of New Decomposition Slice (N)

If the DS-Cv_i and DS-Mv_i exist and there are no differences between them, then the DS-Cv_i and the DS-Mv_i will be included in a set of pairs of Similar Decomposition Slice (S). This means that variable v_i exists in both C and M and the DS-Cv_i and the DS-Mv_i are similar. If both DS-Cv_i and DS-Mv_i are different, then both decomposition slices will be included in a set of pairs of Difference Decomposition Slice (D). This means that variable v_i exists in both C and M, but the DS-Cv_i and the DS-Mv_i have some differences. The differences can be where the statements in the DS-Cv_i have been changed, deleted or there are added statements.

For cases where DS-Cv_i and DS-Mv_i exist, the ReTSE model use the *diff* tool which is a file comparison tool that highlights the differences between two files. The *diff* tool is capable of identifying which lines in the two programs have

changed, been deleted or have an added statement. Therefore, the *diff* tool can help in the first part of the Comparison Phase to identify whether DS-Cv_i and DS-Mv_i are included in the sets of S or D. Both DS-Cv_i and DS-Mv_i are included in the S if there is no output from the *diff* tool. Otherwise, both DS-Cv_i and DS-Mv_i are included in D if there is an output from the *diff* tool.

If variable v_i only exists in C but not in M then DS-Cv_i will be included in the set Delete Decomposition Slice (L). This means there is a decomposition slice for variable v_i in C, but not in M. Otherwise if variable v_i only exists in M but not in C then DS-Mv_i will be included in the set New Decomposition Slice (N). This means there is a decomposition slice for variable v_i in M but not in C. A summary of the first part of the Comparison Phase is shown in the following notations:

- Set of Similar Decomposition Slices (S)

  S = {(DS-Cv_i , DS-Mv_i) | i = 1, 2,... v_c & DS-Cv_i = DS-Mv_i}

- Set of Difference Decomposition Slices (D)

  D = {(DS-Cv_i , DS-Mv_i) | i = 1, 2,... v_c & DS-Cv_i ≠ DS-Mv_i}

- Set of Delete Decomposition Slices (L)

  L = {DS-Cv_i | v_i exists in C & v_i does not exist in M}

- Set of New Decomposition Slices (N)

  N = {DS-Mv_i | v_i exists in M & v_i does not exist in C}

The second part of the Comparison Phase is a more detailed comparison between DS-Cv_i and DS-Mv_i only if they are members of D by analysing the

output from the *diff* tool. The *diff* tool will produce three different types of output; change, delete, or add, as shown in the following notations:

(i) Change
n1,n2**c**n3,n4
<old line (from DS-Cv_i)
- - -
>new line (from DS-Mv_i)

(ii) Delete
n1,n2**d**n3
<old line (from DS-Cv_i)

(iii) Add
n1**a**n3,n4
>new line (from DS-Mv_i)

The meaning of the first type (change) is to replace all old lines specified in the range n1 to n2 from DS-Cv_i with all new lines specified in the range n3 to n4 from DS-Mv_i. The **c** character in the output stands for change. Any statement in the range n1 to n2 from DS-Cv_i will be included in the set of Change Statements for v_i (CSv_i). Any statement in the range n3 to n4 from DS-Mv_i will be included in a set of Relevant Change Statements for v_i (RCSv_i). If the statements in the range n3 to n4 are located at any branch of an *if* statement, whether a *true* or a *false* branch, then all other statements in the same branch will be included in RCSv_i. If the statements in the range n3 to n4 are located at any body of looping statements (e.g., *while* and *for*), then all other statements in the same body will be included in RCSv_i.

The meaning of the second type (delete) is to remove all lines specified in the range n1 to n2 from DS-Cv_i immediately after line n3 from DS-Mv_i. The **d**

character in the output stands for delete. Any statement within the range of n1 to n2 from DS-Cv_i will be included in a set of Change Statements for v_i (CSv_i). A statement at line n3 from DS-Mv_i will be included in a set of Relevant Change Statements for v_i (RCSv_i). If line n3 is not a statement, then the statement immediately after the line n3 will be included in the RCSv_i. This special rule is designed to accommodate the way the diff tool works. In this case, line n3 can be:

- An open curly bracket ({) or a close curly bracket (}) of any branch of an *if* statement.
- An open curly bracket ({) or a close curly bracket (}) of the body of a *while* or a *for* statement.

The meaning of the third type (add) is to add all lines specified in the range of n3 to n4 from DS-Mv_i immediately after line n1 from DS-Cv_i. The **a** character in the output stands for add. Any statement within the range of n3 to n4 from DS-Mv_i will be included in a set of Relevant Change Statements for v_i (RCSv_i). If the statements in the range n3 to n4 are located at any branch of an *if* statement, either a *true* or a *false* branch, then all other statements in the same branch will be also included in the RCSv_i. If the statements in the range n3 to n4 are located at any body of looping statements (e.g., *while* and *for*), then all other statements in the same body will be included in RCSv_i. A statement at n1 from DS-Cv_i will be included in a set of Change Statements for v_i (CSv_i). If line n1 is not a statement, then the statement immediately after line n1 will be included in the CSv_i. In this case, line n1 can be:

- An open curly bracket ({) or a close curly bracket (}) of any branch of an *if* statement.
- An open curly bracket ({) or a close curly bracket (}) of the body of a *while* or a *for* statement.

The union of all CSv_i will produce a set of Change Statements (CS). The union of all RCSv_i will produce a set of Relevant Change Nodes (RCS).

5.2.3 Phase 3: Exclusion

Input:

- Set of Change Statements (CS)
- Test Suite (TS):
  - Test Case (TC)
  - Test History (TH)

Output:

- Set of Excluded Tests (ET)
- Set of Regression Tests (RT)

There are two inputs in the Exclusion Phase. One of them is from the outputs of the Comparison Phase which are a set of Change Statements (CS). The other input is an existing Test Suite (TS) of the Certified Program (C). The TS includes a set of pairs of Test Case (TC) and its Test History (TH). The TC is a set of test cases for the Certified Program (C). The TH is a set of test histories of TC

where TH_i is a set of statements executed for the particular TC_i. The TC_i will be excluded from the TS if CS is not subset of TH_i. This TC_i is included in a set of Excluded Tests (ET). ET is a set of test cases that do not need to be used for regression testing. The remaining test cases in the TS will be the set of Regression Tests (RT). A summary of this Exclusion Phase is shown in the following notations:

$$TS = \{(TC_i, TH_i) \mid \forall i\}$$

$$TC = \{TC_i \mid \forall i\}$$

$$TH = \{TH_i \mid \forall i\}$$

Therefore,

$$ET = \{TC_i \mid ((CS \not\subset TH_i \ \& \ CS \neq \{\}) \text{ or } (CS = \{\})) \ \forall i\}$$

$$RT = \{TC_i \mid \forall i\} \text{ - } ET$$

If CS is an empty set then all TC_i will be excluded from the TS. It means that all TC_i will be included in the ET. Therefore, there is no TC_i in the RT. The CS can be an empty set when D is an empty set. This means M is only involved in adding or deleting variables that lead to an increase in a member of the L or N or both of them.

5.2.4 Phase 4: Optimisation

Input:

- Set of pairs of Difference Decomposition Slices (D)
- Set of New Decomposition Slices (N)

- Set of Regression Tests (RT)
- Set of Relevant Change Statements (RCS)

Output:

- Set of Optimised Regression Tests (RTO)
- Request for New Test Cases

There are four inputs in the Optimisation Phase. Three of them are from the outputs of the Comparison Phase which are a set of pairs of Difference Decomposition Slices (D), a set of New Decomposition Slices (N) and a set of Relevant Change Statements (RCS). The last input is from the output of the Exclusion Phase which is a set of Regression Tests (RT).

The main objective of this phase is to produce a set of Optimised Regression Tests (RTO) because the RT that has been produced by the Exclusion Phase probably has redundant test cases. In some cases, the model will request new additional test cases. These can be achieved by using the algorithm shown in Figure 5.4. The algorithm will execute all test cases which are members of RT onto the union of all DS-Mv_i where DS-Mv_i is a member of D. This union is called UDS-M. RTE$_i$ is a set of executed statements constructed by running TC$_i$ (member of RT) onto UDS-M. Then, RTE$_i$ will be mapped onto RCS. RCS-current is a set of statements in RTE$_i$ that has covered the elements of RCS. In other words, RCS-current is the intersection of RTE$_i$ and RCS. RCS-coverage is a set of statements that are union between previous RCS-coverage and RCS-current.

If the set RCS-current is a subset of RCS-coverage then the next TC_i will be executed. This situation means that the current TC_i is ignored for RTO because it's RTE_i has covered the elements of RCS that is the same or less than the coverage from the previous test cases. If RCS-current is a superset or equal to RCS-coverage then the current TC_i is only assigned to the RTO while at the same time removing all existing TC_i from the RTO. Then, RCS-current is set to RCS-coverage. If RCS-current is not a subset of the set of the current RCS-coverage then the current TC_i should be added to the RTO, and the RCS-current should be added to the set of RCS-coverage. The execution of test cases will stop when the RCS obtains full coverage. The RCS obtains full coverage when RCS is equal to RCS-coverage. Then, the remaining test cases in RT will be ignored for RTO.

If RCS has failed to achieve full coverage after the execution of all TC_i (members of RT) onto UDS-M, then the model will flag to the user that additional new test cases are needed. If N is not an empty set, then the model will also flag to a user that additional new test cases are needed to cover all decomposition slices in the N. The empty set N means that there are no decomposition slices in N. However, the process of designing the additional new test cases is beyond the scope of the ReTSE model. The summary of this Optimisation Phase is shown in the algorithm in Figure 5.4 where:

- UDS-M : the union of DS-Mv_i where DS-Mv_i are members of the D.
- UDS-M $= \cup$ DS-Mv_i | DS-M$v_i \in$ D.
- RTE_i : a set of statements that executed by running $TC_i \in$ RT onto UDS-

M.

- RCS-current : a set of statements.
- RCS-coverage : a set of statements.

The final results of applying this model are:

(i) RTO - a set of optimised regression tests.

(ii) Request_New_Test_Cases - Indicating a request for new test cases.

```
Begin
  RTO = null;
  RCS-current = null;
  RCS-coverage = null;
  RCS-full = NO;
  Request_New_Test_Cases = NO;
  While (TCi ∈ RT && RCS-full == NO) do
  {
     Construct RTEi;
     RCS-current = RCS ∩ RTEi;
     If (RCS-current ⊂ RCS-coverage) then
         Next;
     Else if (RCS-current ⊇ RCS-coverage) then
     {
         RTO = {TCi};
         RCS-coverage = RCS-current;
     }
     Else // if (RCS-current not subset RCS-coverage)
     {
         RTO = RTO ∪ TCi;
         RCS-coverage = RCS-coverage ∪ RCS-current;
     }
     If (RCS == RCS-coverage) then
         RCS-full = YES;
  }
  If (RT == null) then
     RTO = null;
  Else if (RCS-full == NO) then
     Request_New_Test_Cases = YES;
  If (N != null) then
     Request_New_Test_Cases = YES;
End
```

Figure 5.4: Optimisation Algorithm

5.3 An Illustration of the ReTSE Model

A sample program (Tax Program) is used to illustrate the ReTSE model. The Tax Program is a simplified version of the original Tax Program that has been used by Danicic et al. [24] and Hierons et al. [57]. The simplified version only has a few conditions to calculate a tax and decide its code compared to the original one. The Tax Program shown in Figure 5.5 is the Original Certified Program (OC) in the model. The program takes two inputs which are income and age, and produces two outputs which are a total of tax that needs to be paid and its relevant tax code. The calculation of tax at statement S8 of the program in Figure 5.5 has a minor change. The new version of the Tax Program called the Original Modified Program (OM) in the model is shown in Figure 5.6. The ReTSE is used in order to obtain optimised regression tests and to identify the requirements for new test cases for the OM.

5.3.1 Phase 1: Program Analysis

In the Program Analysis Phase, both the original certified and original modified programs of the Tax Program face two steps which are Pretty Print and Slicing.

5.3.1.1 Step 1.1: Pretty Print

In Pretty Print, the Original Certified Program (OC) in Figure 5.5 and Original Modified Program (OM) in Figure 5.6 will be transformed into a standard format of programming style as shown in Figure 5.7 and Figure 5.8 respectively. The programs in Figure 5.7 and Figure 5.8 are called Certified Program (C) and Modified Program (M) respectively. A declaration in statement S1 of the program

```
   #include <stdio.h>
   main()
   {
S1    int income, tax, age;
S2    char code;
S3    scanf("%d", &income);
S4    scanf("%d", &age);
S5    if (income < 10000)
S6    tax = 0;
      else {
S7    income = income - 10000;
S8    tax = (income*40/100); //old
      }
S9    if (age < 65)
S10   code = 'L';
S11   else if (age < 75)
S12   code = 'P';
      else
S13   code = 'T';
S14   printf("%d\n", tax);
S15   printf("%c\n", code);
   }
```

Figure 5.5: Original Certified Program (OC)

in Figure 5.5 and S1' of the program in Figure 5.6 is decomposed into individual declarations of every variable as shown in statements S1, S2 and S3 of the program in Figure 5.7 and statements S1', S2' and S3' of the program in Figure 5.8. A comment in statements S8 and S8' of the programs in Figure 5.5 and Figure 5.6 respectively are removed from the programs. Every branch of *if* statement has its own curly bracket ({}) even when it has only one statement as shown in statement S8 in Figure 5.7.

```
    #include <stdio.h>
    main()
    {
S1’     int income, tax, age;
S2’     char code;
S3’     scanf("%d", &income);
S4’     scanf("%d", &age);
S5’     if (income < 10000)
S6’     tax = 0;
        else {
S7’       income = income - 10000;
S8’       tax = (income*30/100); //new
        }
S9’     if (age < 65)
S10’    code = ’L’;
S11’    else if (age < 75)
S12’    code = ’P’;
        else
S13’    code = ’T’;
S14’    printf("%d\n", tax);
S15’    printf("%c\n", code);
    }
```

Figure 5.6: Original Modified Program (OM)

5.3.1.2 Step 1.2: Slicing

In Slicing, both the Certified Program (C) and the Modified Program (M) in Figure 5.7 and Figure 5.8 respectively have been sliced using the decomposition slicing technique. The decomposition slice is built for one variable and is the union of the slices taken at the line numbers of the use of the given variable. For instance, the decomposition slice for variable *income* in Figure 5.10(a) is produced from the union of three backward slices of variable *income* taken from statements where it is used. The uses of variable *income* are in statements S7,

```
   #include <stdio.h>
   main()
   {
S1   int income;
S2   int tax;
S3   int age;
S4   char code;
S5   scanf("%d", &income);
S6   scanf("%d", &age);
S7   if (income < 10000)
     {
S8      tax = 0;
     }
     else
     {
S9      income = income - 10000;
S10     tax = (income*40/100);
     }
S11  if (age < 65)
     {
S12     code = 'L';
     }
S13  else if (age < 75)
     {
S14     code = 'P';
     }
     else
     {
S15     code = 'T';
     }
S16  printf("%d\n", tax);
S17  printf("%c\n", code);
   }
```

Figure 5.7: Certified Program (C)

```
    #include <stdio.h>
    main()
    {
S1’   int income;
S2’   int tax;
S3’   int age;
S4’   char code;
S5’   scanf("%d", &income);
S6’   scanf("%d", &age);
S7’   if (income < 10000)
      {
S8’      tax = 0;
      }
      else
      {
S9’      income = income - 10000;
S10’     tax = (income*30/100);
      }
S11’  if (age < 65)
      {
S12’     code = ’L’;
      }
S13’  else if (age < 75)
      {
S14’     code = ’P’;
      }
      else
      {
S15’     code = ’T’;
      }
S16’  printf("%d\n", tax);
S17’  printf("%c\n", code);
    }
```

Figure 5.8: Modified Program (M)

S9 and S10. These backward slices are shown in Figure 5.9.

There are four decomposition slices for each C and M corresponding to four variables which are *income*, *tax*, *age* and *code*. Decomposition slices for C (DS-C) are shown in Figure 5.10 - 5.13 in part (a) and decomposition slices for M (DS-M) are shown in part (b) of the same figures. The output summary of the Slicing Step is shown below:

- DS-C = {DS-C$_{income}$, DS-C$_{tax}$, DS-C$_{age}$, DS-C$_{code}$}
- DS-M = {DS-M$_{income}$, DS-M$_{tax}$, DS-M$_{age}$, DS-M$_{code}$}

5.3.2 Phase 2: Comparison

There are two parts in the Comparison Phase. Firstly, the decomposition slices in the DS-C are compared to the decomposition slices in the DS-M using the *diff* tool. For instance, the DS-C$_{income}$ in Figure 5.10(a) is compared to the DS-M$_{income}$ in Figure 5.10(b). The output produced from the *diff* tool shows that both decomposition slices (DS-C$_{income}$ and DS-M$_{income}$) are not the same. Therefore both decomposition slices are included in a set of pairs of Difference Decomposition Slice (D). The DS-C$_{tax}$ and DS-M$_{tax}$ (Figure 5.11) are also included in D because there is an output produced from the *diff* tool for this comparison. The comparisons between DS-C$_{age}$ and DS-M$_{age}$ (Figure 5.12) and DS-C$_{code}$ and DS-M$_{code}$ (Figure 5.13) do not produce any output from the *diff* tool. Therefore, those decomposition slices are included in a set of pairs of Similar Decomposition Slice (S). There is no decomposition slice included in set N and L. The output summary of the first part of the Comparison Phase is shown in Table 5.1.

```
  #include <stdio.h>
  main()
  {
S1  int income;
S5  scanf("%d", &income);
S7  if (income < 10000)
    else
  }
```

(a) The use at S7

```
  #include <stdio.h>
  main()
  {
S1  int income;
S5  scanf("%d", &income);
S7  if (income < 10000)
    else
    {
S9    income=income-10000;
    }
  }
```

(b) The use at S9

```
  #include <stdio.h>
  main()
  {
S1  int income;
S2  int tax;
S5  scanf("%d", &income);
S7  if (income < 10000)
    else
    {
S9    income=income-10000;
S10   tax=(income*40/100);
    }
  }
```

(c) The use at S10

Figure 5.9: Backward Slices for Variable *income* at its Uses

Table 5.1: Comparison Results between DS-Cv_i and DS-Mv_i

Set of	Member of Set
D	{(DS-C$_{income}$, DS-M$_{income}$), (DS-C$_{tax}$, DS-M$_{tax}$)}
S	{(DS-C$_{age}$, DS-M$_{age}$), (DS-C$_{code}$, DS-M$_{code}$)}
L	{}
N	{}

```
[L1]#include <stdio.h>
    main()
    {
S1    int income;
S2    int tax;
S5    scanf("%d", &income);
S7    if (income < 10000)
      {
      }
[10]  else
      {
S9      income=income-10000;
S10     tax=(income*40/100);
      }
[15]}
```

(a) DS-C$_{income}$

```
[L1]#include <stdio.h>
    main()
    {
S1'   int income;
S2'   int tax;
S5'   scanf("%d", &income);
S7'   if (income < 10000)
      {
      }
[10]  else
      {
S9'     income=income-10000;
S10'    tax=(income*30/100);
      }
[15]}
```

(b) DS-M$_{income}$

Figure 5.10: Decomposition Slice for Variable *income*

```
[L1]#include <stdio.h>
    main()
    {
S1    int income;
S2    int tax;
S5    scanf("%d", &income);
S7    if (income < 10000)
      {
S8      tax = 0;
[L10  }
      else
      {
S9      income=income-10000;
S10     tax=(income*40/100);
      }
S16   printf("%d\n", tax);
    }
```

(a) DS-C$_{tax}$

```
[L1]#include <stdio.h>
    main()
    {
S1'   int income;
S2'   int tax;
S5'   scanf("%d", &income);
S7'   if (income < 10000)
      {
S8'     tax = 0;
[L10] }
      else
      {
S9'     income=income-10000;
S10'    tax=(income*30/100);
      }
S16'  printf("%d\n", tax);
    }
```

(b) DS-M$_{tax}$

Figure 5.11: Decomposition Slice for Variable *tax*

```
    #include <stdio.h>
    main()
    {
S3    int age;
S6    scanf("%d", &age);
S11   if (age < 65)
      {
      }
S13   else if (age < 75)
      {
      }
      else
      {
      }
    }
```

(a) DS-C$_{age}$

```
    #include <stdio.h>
    main()
    {
S3'   int age;
S6'   scanf("%d", &age);
S11'  if (age < 65)
      {
      }
S13'  else if (age < 75)
      {
      }
      else
      {
      }
    }
```

(b) DS-M$_{age}$

Figure 5.12: Decomposition Slice for Variable *age*

```
    #include <stdio.h>
    main()
    {
S3    int age;
S4    char code;
S6    scanf("%d", &age);
S11   if (age < 65)
      {
S12     code = 'L';
      }
S13   else if (age < 75)
      {
S14     code = 'P';
      }
      else
      {
S15     code = 'T';
      }
S17   printf("%c\n", code);
    }
```

(a) DS-C$_{code}$

```
    #include <stdio.h>
    main()
    {
S3'   int age;
S4'   char code;
S6'   scanf("%d", &age);
S11'  if (age < 65)
      {
S12'    code = 'L';
      }
S13'  else if (age < 75)
      {
S14'    code = 'P';
      }
      else
      {
S15'    code = 'T';
      }
S17'  printf("%c\n", code);
    }
```

(b) DS-M$_{code}$

Figure 5.13: Decomposition Slice for Variable *code*

The second part of the Comparison Phase is a more detailed comparison between DS-Cv_i and DS-Mv_i only if they are members of D. It analyses the output from the *diff* tool. In the given example, only decomposition slices of variable *income* (DS-C$_{income}$, DS-M$_{income}$) and *tax* (DS-C$_{tax}$, DS-M$_{tax}$) are involved in the second part of the comparison because they are members of the D that resulted in the first part of the comparison. The comparison output using the *diff* tool as below:

- Comparison between DS-C$_{income}$ and DS-M$_{income}$

```
13c13
<    tax = (income*40/100);
- - -
>    tax = (income*30/100);
```

- Comparison between DS-C$_{tax}$ and DS-M$_{tax}$

```
14c14
<    tax = (income*40/100);
- - -
>    tax = (income*30/100);
```

In the comparison between DS-C$_{income}$ and DS-M$_{income}$ (Figure 5.10), the statement at line 13 ([L13]) from DS-C$_{income}$ is included in the set of Change Statements for variable *income* (CS$_{income}$). Any statement at L13 from DS-M$_{income}$ is included in the set of Relevant Change Statements for variable *income* (RCS$_{income}$). Therefore, the statement S10 from DS-C$_{income}$ is included in the CS$_{income}$ and the statement S10' from DS-M$_{income}$ is included in the RCS$_{income}$. Statement S9' is also included in the RCS$_{income}$ because it is located at the same branch of statement S10'.

In the comparison between DS-C_{tax} and DS-M_{tax} (Figure 5.11), the statement at line 14 ([L14]) from DS-C_{tax} is included in the set of Change Statements for variable *tax* (CS_{tax}). Any statement at L14 from DS-M_{tax} is included in the set of Relevant Change Statements for variable *tax* (RCS_{tax}). Therefore, the statement S10 from DS-C_{tax} is included in the CS_{tax} and the statement S10' from DS-M_{tax} is included in the RCS_{tax}. Statement S9' is also included in the RCS_{tax} because it is located at the same branch of statement S10'. Then the CS is produced from the union of CS_{income} and CS_{tax} where the RCS is produced from the union of RCS_{income} and RCS_{tax}. A summary of the second part of the Comparison Phase is shown below:

$$\begin{aligned} CS &= CS_{income} \cup CS_{tax} \\ &= \{S10\} \cup \{S10\} \\ &= \{S10\} \end{aligned}$$

$$\begin{aligned} RCS &= RCS_{income} \cup RCS_{tax} \\ &= \{S9', S10'\} \cup \{S9', S10'\} \\ &= \{S9', S10'\} \end{aligned}$$

5.3.3 Phase 3: Exclusion

There are six test cases in the existing test suite of the Certified Program of the Tax Program as shown in Table 5.2. Every Test Case (TC_i) in the Test Suite (TS) has its own coverage onto the Certified Program called Test History (TH_i). It has been designed using all path coverage of the program [33]. The TH_i is shown in Table 5.3. Symbol "X" and "-" indicate statements executed and not executed

for a particular TC_i. The CS that has been produced from the Comparison Phase has only statement S10. Any TC_i where the CS is not subset of TH_i then the TC_i will be included in the set of Excluded Test (ET). In this example, the statement S10 is not subset of TH_1, TH_2 and TH_3. Therefore, TC_1, TC_2, and TC_3 will be included in the set of ET. The remaining test cases in Test Suite are included in the set of Regression Tests (RT). Therefore, TC_4, TC_5 and TC_6 are included in the RT. The output summary of this phase is shown below:

RT = {TC_4, TC_5, TC_6}

Table 5.2: Test Suite for Certified Program (Tax Program)

Test Case (TC_i)	Input	Output
TC_1	income = 9000 age = 50	0 L
TC_2	income = 9000 age = 70	0 P
TC_3	income = 9000 age = 75	0 T
TC_4	income = 12000 age = 50	800 L
TC_5	income = 12000 age = 70	800 P
TC_6	income = 12000 age = 75	800 T

5.3.4 Phase 4: Optimisation

In the Optimisation Phase, all TC_i that are members of the RT will be executed onto UDS-M. In this example, there are only two DS-Mv_i members of D which are DS-M_{income} and DS-M_{tax}. The union of both decomposition slices (UDS-M) will

Table 5.3: Test History (TH_i) of TC_i for Certified Program

Statement	TH_1	TH_2	TH_3	TH_4	TH_5	TH_6
S1	X	X	X	X	X	X
S2	X	X	X	X	X	X
S3	X	X	X	X	X	X
S4	X	X	X	X	X	X
S5	X	X	X	X	X	X
S6	X	X	X	X	X	X
S7	X	X	X	X	X	X
S8	X	X	X	-	-	-
S9	-	-	-	X	X	X
S10	-	-	-	X	X	X
S11	X	X	X	X	X	X
S12	X	-	-	X	-	-
S13	-	X	X	-	X	X
S14	-	X	-	-	X	-
S15	-	-	X	-	-	X
S16	X	X	X	X	X	X
S17	X	X	X	X	X	X

be the same slice as DS-M_{tax} as shown in Figure 5.11(b) which includes statements S1', S2', S5' S7', S8', S9', S10' and S16'. The RCS produced in the second part of the Comparison Phase is used here. The RCS includes statements S9' and S10'.

The test cases in RT, which are TC_4, TC_5 and TC_6, are sequently executed onto the UDS-M. Firstly, the TC_4 is executed onto UDS-M. The RTE_4 for TC_4 is S1', S2', S5' S7', S9', S10' and S16'. The intersection between RTE_4 and RCS contains all members of RCS. That means the RCS obtains full coverage by executed only the TC_4. The execution of test cases is stopped because the RCS has already achieved full coverage. This means that it is enough to use only TC_4 as a regression test for the modified program. Therefore, TC_4 will be included in the RTO. The remaining test cases TC_5 and TC_6 in RT are ignored for RTO. A

summary of the Optimisation Phase is given below:

UDS-M = {S1', S2', S5' S7', S8', S9', S10', S16'}

RCS = {S9', S10'}

TC_4

RTE_4 = {S1', S2', S5' S7', S9', S10', S16'}

RCS-current = {S9', S10'}

RCS-coverage = {S9', S10'}

RTO = $\{TC_4\}$

RCS-full = YES

The final output of the model for this example is given below:

RTO = $\{TC_4\}$

Request_New_Test_Cases = NO

5.4 Summary

This chapter has discussed the ReTSE model. The model is explained sequentially phase by phase. There are four main phases which have been proposed in the model which are Program Analysis (which includes Pretty Print and Slicing steps), Comparison, Exclusion and Optimisation. Then, the model is illustrated by using a small program as an example. The results of this example show that the ReTSE model works for that program. Moreover, the model has reduced three test cases from six test cases in TS at the Exclusion Phase which is 50% reduction. The model once again has reduced another two test cases from RT at

the Optimisation Phase. More case studies are presented in Chapter 7.

Chapter 6

Implementation

6.1 Introduction

This chapter shows how the ReTSE model can be implemented as a fully automated and integrated tool. The chapter is organised as follows. The following section discusses existing tools that were used and adapted in the model, through their relevant phases. The section also discusses how to fully implement the model in the future.

6.2 Current and Future Implementation

This section discusses the current and future implementation of the ReTSE model. There are some existing tools that are used in the current prototype implementation. Each tool is described in its relevant phases. Figure 6.1 shows a sequential dataflow diagram of the ReTSE model. Every phase has it own data input and output. The original inputs are Original Certified (OC), Original Modified (OM)

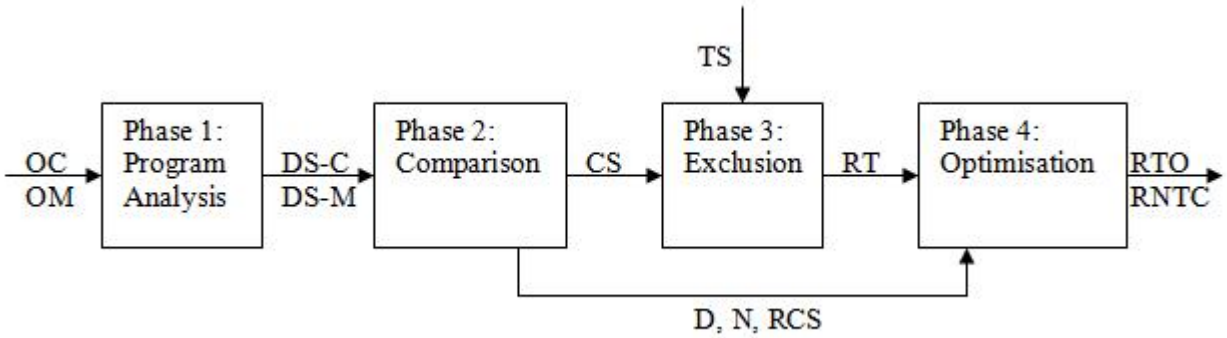

Figure 6.1: A Sequential Dataflow Diagram of the ReTSE Model

and Test Suite (TS). The final outputs are a set of an Optimised Regression Tests (RTO) and a Request for New Test Cases (RNTC).

6.2.1 Phase 1: Program Analysis

6.2.1.1 Step 1.1: Pretty Print

Currently, the Pretty Print Step is done manually. However, there is a number of existing tools that have similar functionality with this step. One of them is a Linux based tool called *indent* [19]. It can be applied to the Pretty Print Step to help the ReTSE model handles large scale programs.

6.2.1.2 Step 1.2: Slicing

The Code Surfer (*csurf*) tool [5] was used in the prototype in order to produce decomposition slices for the Certified Program (C) and the Modified Program (M). Generally, Code Surfer is a program understanding tool that makes manual re-

viewing code easier and faster. CodeSurfer calculates a variety of representations that can be explored through the graphical user interface or accessed through the Application Programming Interface (API).

Decomposition slices are produced from the union of slices taken at the uses of a variable. These slices are computed by the *csurf* tool. For example, the decomposition slice for the variable *income* in Figure 5.10 (Chapter 5, page 80) is computed using the *csurf* tool. There are three uses of the variable *income* in the program in Figure 5.7 (Chapter 5, page 76). These uses are located at statements S7, S9 and S10. Every slice of these uses can be computed with the *csurf* tool as shown in Figure 6.2, Figure 6.3 and Figure 6.4 respectively. The slices are computed using the backward slicing option in the tool and highlighted in red. The uses of the variable *income* are highlighted in yellow as shown in those figures. Then, the decomposition slice for the variable *income* is computed by combining these slices. The decomposition slice for the variable *income* is shown in Figure 6.5.

Currently, the *csurf* tool has only been used in the Slicing Step of the ReTSE model. However, it seems that the tool can be used throughout the model in future. This means that the ReTSE model can become part of the csurf tool. This is based on the fact that the tool can be programmed, extended, customised and integrated with other applications using its scripting language. This scripting language is based on Schema, a general purpose programming language.

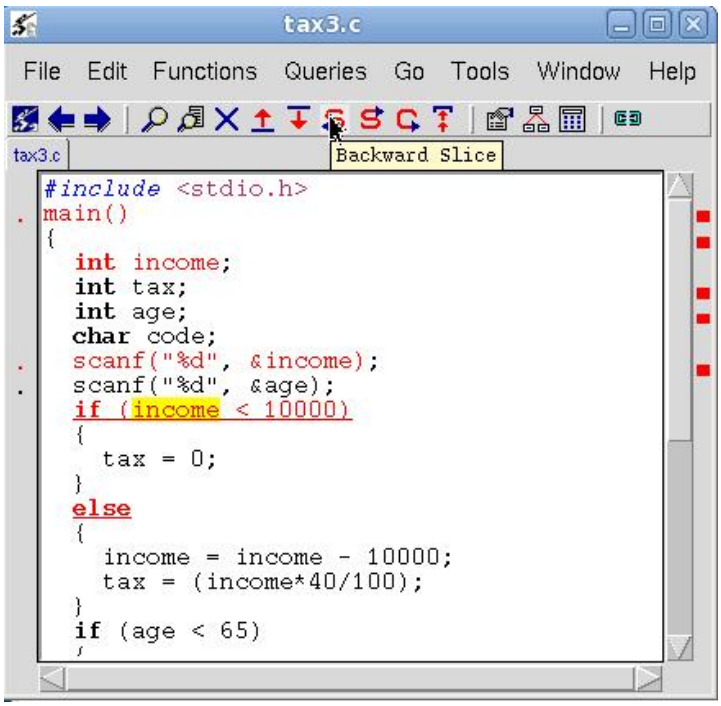

Figure 6.2: Backward Slice for Variable *income* at Use 1

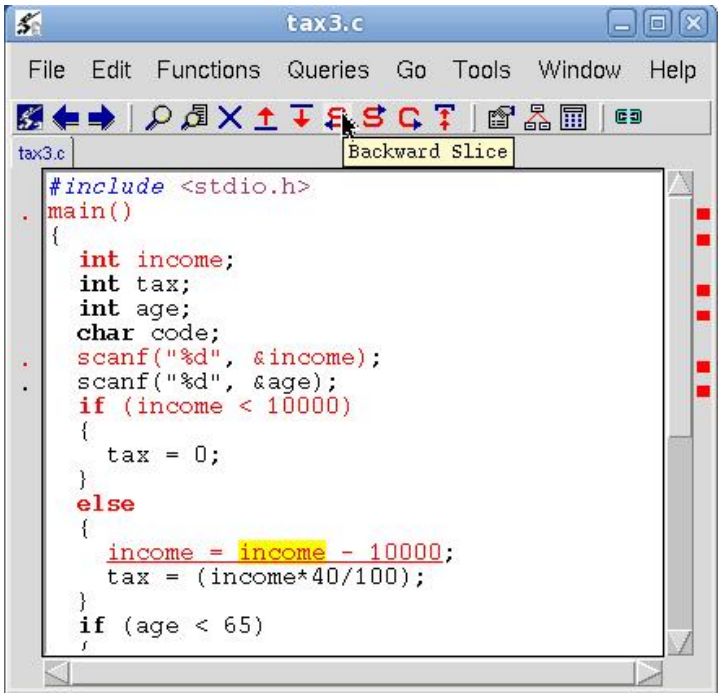

Figure 6.3: Backward Slice for Variable *income* at Use 2

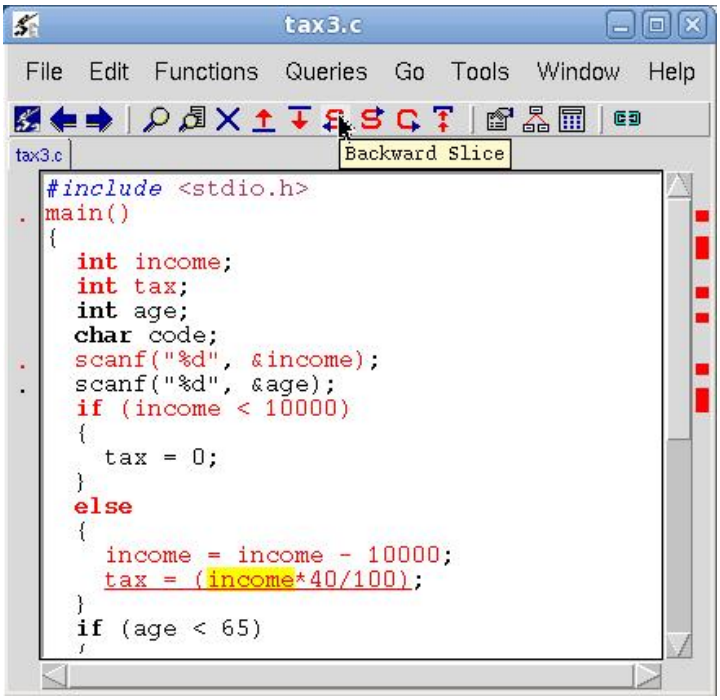

Figure 6.4: Backward Slice for Variable *income* at Use 3

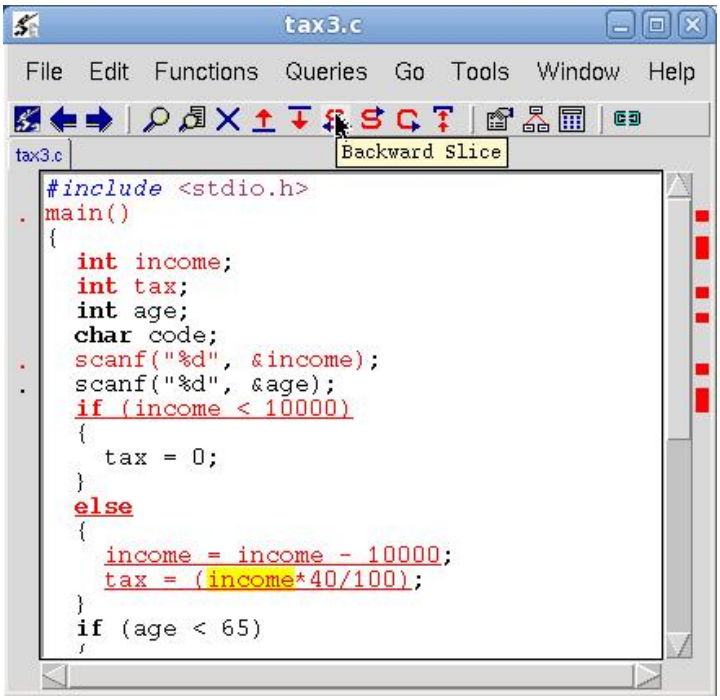

Figure 6.5: Decomposition Slice for Variable *income*

6.2.2 Phase 2: Comparison

The *diff* tool [26] was used in both parts of the Comparison Phase. In the first part, the *diff* tool was only used for a general comparison between the two decomposition slices taken from DS-C and DS-M. The tool was only used for cases where DS-Cv_i and DS-Mv_i existed in DS-C and DS-M respectively. This comparison can produce a set of pairs of Similar Decomposition Slices (S) or a set of pairs of Difference Decomposition Slices (D). However, the production of a set of Delete Decomposition Slices (L) and a set of New Decomposition Slices (N) is still performed manually. In the future, this can be implemented by writing code that can identify which decomposition slices only exist in DS-C or DS-M.

In the second part of the Comparison Phase, the *diff* tool was used intensively. The comparison is specifically for the decomposition slices which are members of D. This part analyses the output from the *diff* tool when comparing the two decomposition slices from D. The *diff* tool shows which lines in the two programs have changed, been deleted or have an added statement.The analysis of this output is presented in Chapter 5 (Section 5.2.2, page 62).

However, the way the ReTSE model matches those lines that are produced from the *diff* tool to the relevant statements in the slices is still done manually. In the future, this can be implemented by writing code that can identify which statements in the slice are related to the selected lines of the program. This code should also be capable of producing a set of Change Statements (CS) and Relevant Change Statements (RCS). The methods to produce CS and RCS are

discussed in Chapter 5 (Section 5.2.2, page 62).

6.2.3 Phase 3: Exclusion

The Exclusion Phase is illustrated in the set notation as shown in Chapter 5 (Section 5.2.3, page 67). The implementation of the Exclusion Phase is based on two inputs which are CS and Test Suite (TS). The TS consists of the Test Case (TC) and its Test History (TH). The calculation of these inputs will produce two outputs which are a set of Excluded Tests (ET) and a set of Regression Tests (RT).

Currently, this phase is done manually. In actual practice, the Certified Program (C) is assumed to have its own existing test suite which includes test cases and its own test history. However, for the purpose of analysis the ReTSE model, Test Specification Language (TSL) and *gcov* will be discussed as they are the tools that have been used in this phase.

Generally, the TSL tool [27] is used for producing the test cases of a program. TSL generates test frames from a specification file written in the extended Test Specification Language. Then, test frames are used as a guideline to produce test cases of the Certified Program (C).

The *gcov* tool is used to generate a test history of each test case. Originally, the tool was used in order to obtain coverage information for each test case. The tool shows which lines in the program are actually executed. However, it is still a manual process in selecting which statements are related to those lines.

Finally, this process can produce a test history which is a set of statements that is executed by a relevant test case.

6.2.4 Phase 4: Optimisation

Currently, the Optimisation Phase is performed manually and no specific tool is used. In Chapter 5, there is an algorithm that can be used to implement the Optimisation Phase. Its implementation is dependent on four inputs which are D, N, RT and RCS that have been produced in previous phases. This phase will produce two outputs which are a set of Optimised Regression Tests (RTO) and a Request for New Test Cases (RNTC).

6.3 Summary

This chapter shows that the ReTSE model can be implemented as a fully automated and integrated tool in the future. The implementation can be divided into four main parts that correspond to the four main phases in the model. There are two phases in the model that have intensively used an existing tool. These are the *csurf* tool in the Slicing Step of the Program Analysis Phase and the *diff* tool in the Comparison Phase. The other phases can be implemented based on set notations or algorithms that have been designed in Chapter 5. The final step is to integrate all these modules to enhance the ReTSE's performance and applicability in future.

Chapter 7

Types of Modification: Case Study

7.1 Introduction

This chapter describes five case studies that correspond to the five types of modifications in order to evaluate the ReTSE model. Another two case studies are used for multiple modifications. The chapter is organised as follow. Five types of modifications are presented in the second section. Each case study illustrates one type of modification at a time. The third section presents another two case studies that have a combination of different types of modifications in the same program.

7.2 Types of Modification

The ReTSE model focuses on five types of modifications. They are:

(i) Change Statements

(ii) Add Statements

(iii) Delete Statements

(iv) Add Variables

(v) Delete Variables

The first three are commonly focused on in existing regression test selection models. The Tax Program as shown in Chapter 5 Figure 5.5 (page 74) is reused to evaluate the model based on these five types of modification. It is assumed that the programs (OC and OM) in the following sections have been through a Pretty Print Step. The same Certified Program (C) in Chapter 5, Figure 5.7 (page 76) is used in the following sections. However, different versions of the Modified Program (M) are used to tackle all the types of modifications.

7.2.1 Modification Type 1 - Change Statements (Case 1)

Change Statements refer to a modification of statements without adding or deleting statements. It can be a change in a static value or a change in a variable used. Application of the model to Change Statements has been described in Chapter 5, Section 5.3 (page 73).

7.2.2 Modification Type 2 - Add Statements (Case 2)

In this case, the M has two additional new statements at S9' and S10' in order to add a new condition for tax calculation.

7.2.2.1 Phase 1: Program Analysis

Step 1.1: Pretty Print

The Pretty Print Step has produced a Certified Program (C) and Modified Program (M) as shown in Chapter 5, Figure 5.7 (page 76) and Figure 7.1 respectively.

Step 1.2: Slicing

In the Slicing Step, both C and M are decomposed into decomposition slices corresponding to the variables in the programs. Therefore, both programs have four decomposition slices. Two decomposition slices for C are shown in part (a) of Figure 7.2 and Figure 7.3 while the two decomposition slices for M are shown in part (b) of the same figures. These decomposition slices are DS-C_{income} and DS-C_{tax} for C and DS-M_{income} and DS-M_{tax} for M that correspond to *income* and *tax* variables. Another two decomposition slices for both C and M are DS-C_{age} and DS-C_{code} for C and DS-M_{age} and DS-M_{code} for M that correspond to *age* and *code* variables, similar to the example mentioned in Chapter 5, Section 5.3 (page 73). These decomposition slices are not shown in this section because there are no differences between them, and are also not needed in the next steps. The output summary of the Slicing Step is given below:

- DS-C = {DS-C$_{income}$, DS-C$_{tax}$, DS-C$_{age}$, DS-C$_{code}$}
- DS-M = {DS-M$_{income}$, DS-M$_{tax}$, DS-M$_{age}$, DS-M$_{code}$}

7.2.2.2 Phase 2: Comparison

In the first part of the Comparison Phase, the decomposition slices in the DS-C are compared to the decomposition slices in the DS-M using the *diff* tool. The DS-C$_{income}$ in Figure 7.2(a) is compared to the DS-M$_{income}$ in Figure 7.2(b). An output is produced from the *diff* tool as a result of this comparison, and thus both decomposition slices are included in a set of pairs of Difference Decomposition Slice (D). The DS-C$_{tax}$ and DS-M$_{tax}$ (Figure 7.3) are also included in D because there is an output produced from the *diff* tool. The comparisons between DS-C$_{age}$ and DS-M$_{age}$ and DS-C$_{code}$ and DS-M$_{code}$ are similar to the one discussed in Chapter 5, Section 5.3.2 (page 78). These decomposition slices are included in the set of pairs of Similar Decomposition Slices (S) because they do not produce any output from the *diff* tool. There are no decomposition slice included in L and N. The output summary of the first part of the Comparison Phase is shown in Table 7.1.

Table 7.1: Comparison Results between DS-Cv_i and DS-Mv_i (Case 2)

Set of	Member of Set
D	{(DS-C$_{income}$, DS-M$_{income}$), (DS-C$_{tax}$, DS-M$_{tax}$)}
S	{(DS-C$_{age}$, DS-M$_{age}$), (DS-C$_{code}$, DS-M$_{code}$)}
L	{}
N	{}

```
    #include <stdio.h>
    main()
    {
S1’   int income;
S2’   int tax;
S3’   int age;
S4’   char code;
S5’   scanf("%d", &income);
S6’   scanf("%d", &age);
S7’   if (income < 10000)
      {
S8’      tax = 0;
      }
S9’   else if (income < 20000)
      {
S10’     tax = ((income-10000)*25/100);
      }
      else
      {
S11’     income = income - 10000;
S12’     tax = (income*40/100);
      }
S13’  if (age < 65)
      {
S14’     code = ’L’;
      }
S15’  else if (age < 75)
      {
S16’     code = ’P’;
      }
      else
      {
S17’     code = ’T’;
      }
S18’  printf("%d\n", tax);
S19’  printf("%c\n", code);
    }
```

Figure 7.1: Modified Program (M) (Case 2)

```
[L1]#include <stdio.h>
    main()
    {
S1    int income;
S2    int tax;
S5    scanf("%d", &income);
S7    if (income < 10000)
      {
[L9]  }
[L10] else
      {
S9      income=income-10000;
S10     tax=(income*40/100);
      }
[15]}
```

(a) DS-C$_{income}$

```
[L1]#include <stdio.h>
    main()
    {
S1'   int income;
S2'   int tax;
S5'   scanf("%d", &income);
S7'   if (income < 10000)
      {
[L9]  }
S9'     else if (income<20000)
      {
S10'    tax=((income-10000)*25/100);
[L13] }
      else
      {
S11'    income=income-10000;
S12'    tax=(income*30/100);
      }
[L19]}
```

(b) DS-M$_{income}$

Figure 7.2: Decomposition Slice for Variable *income* (Case 2)

```
[L1]#include <stdio.h>
    main()
    {
S1    int income;
S2    int tax;
S5    scanf("%d", &income);
S7    if (income < 10000)
      {
S8      tax = 0;
[L10  }
      else
      {
S9      income=income-10000;
S10     tax=(income*40/100);
      }
S16   printf("%d\n", tax);
    }
```

(a) DS-C$_{tax}$

```
[L1]#include <stdio.h>
    main()
    {
S1'   int income;
S2'   int tax;
S5'   scanf("%d", &income);
S7'   if (income < 10000)
      {
S8'     tax = 0;
[L10] }
S9'   else if (income<20000)
[L12] {
S10'    tax=((income-10000)*25/100);
[L14] }
      else
      {
S11'    income=income-10000;
S12'    tax=(income*30/100);
      }
S18'  printf("%d\n", tax);
    }
```

(b) DS-M$_{tax}$

Figure 7.3: Decomposition Slice for Variable *tax* (Case 2)

The second part of the Comparison Phase is a more detailed comparison between DS-Cv_i and DS-Mv_i only if they are members of D. In this case, only the decomposition slices for variables *tax* (DS-C$_{tax}$ and DS-M$_{tax}$) and *income* (DS-C$_{income}$ and DS-M$_{income}$) are involved in the second part of the comparison because they are members of D. The comparison outputs using the *diff* tool are:

- Comparison between DS-C$_{income}$ and DS-M$_{income}$

```
9a10,13
>   else if (income < 20000)
>   {
>      tax = ((income-10000)*25/100
>   }
```

- Comparison between DS-C$_{tax}$ and DS-M$_{tax}$

```
10a11,14
>   else if (income < 20000)
>   {
>       tax = ((income-10000)*25/100
>   }
```

In the comparison between DS-C$_{income}$ and DS-M$_{income}$ (Figure 7.2), the statement at line 9 ([L9]) from DS-C$_{income}$ is included in the set of Change Statements for variable *income* (CS$_{income}$). The statements in the range from line 10 ([L10]) to line 13 ([l13]) from DS-M$_{income}$ are included in the set of Relevant Change Statements for variable *income* (RCS$_{income}$). Line 9 is not a statement but a closed curly bracket (}), so that, the statement located immediately after line 9 is included in the CS$_{income}$. Therefore, statement S9 from DS-C$_{income}$ is included in the CS$_{income}$ and statements S9' and S10' from DS-C$_{income}$ are included in the

RCS_{income}. Statements S11' and S12' are also included in the RCS_{income} because they are located at the same branch of statement S9'.

As for the comparison between DS-C_{tax} and DS-M_{tax} (Figure 7.3), the statement at line 10 ([L10]) from DS-C_{tax} is included in the set of Change Statements for variable *tax* (CS_{tax}). Any statements in the range from line 11 ([L11]) to line 14 ([L14]) from DS-M_{tax} are included in the set of Relevant Change Statements for variable *tax* (RCS_{tax}). Line 10 is not a statement but a closed curly bracket (}), so that, the statement located immediately after line 10 is included in the CS_{tax}. Therefore, statement S9 from DS-C_{tax} is included in the CS_{tax} and statements S9' and S10' from DS-M_{tax} are included in the RCS_{tax}. Statements S11' and S12' are also included in the RCS_{tax} because they are located at the same branch of statement S9'. The CS is produced from the union of CS_{tax} and CS_{income}, while the RCS is produced from the union of RCS_{tax} and RCS_{income}. A summary of the second part of the Comparison Phase is given below:

$$\begin{aligned} CS &= CS_{income} \cup CS_{tax} \\ &= \{S9\} \cup \{S9\} \\ &= \{S9\} \end{aligned}$$

$$\begin{aligned} RCS &= RCS_{income} \cup RCS_{tax} \\ &= \{S9', S10', S11', S12'\} \cup \{S9', S10', S11', S12'\} \\ &= \{S9', S10', S11', S12'\} \end{aligned}$$

7.2.2.3 Phase 3: Exclusion

The same test cases (TC) and test histories (TH) shown in Table 5.2 (page 84) and Table 5.3 (page 85) in Chapter 5 are used in this phase. Any TC_i where the CS is not a subset of TH_i will be included in the set of Excluded Test (ET). In this example, statement S9, a member of CS is not a subset of TH_1, TH_2 and TH_3. Therefore, TC_1, TC_2, and TC_3 will be included in the ET. The remaining test cases in the Test Suite are included in the set of Regression Tests (RT). Therefore, TC_4, TC_5 and TC_6 are included in the RT. A summary of the Exclusion Phase is given below:

$$RT = \{TC_4, TC_5, TC_6\}$$

7.2.2.4 Phase 4: Optimisation

In the Optimisation Phase, all TC_i that are members of RT will be executed onto UDS-M. UDS-M is the union of all DS-Mv_i where DS-Mv_i is a member of D. In this case study, there are only two DS-Mv_i members of D, DS-M_{income} and DS-M_{tax}. The union of both decomposition slices (UDS-M) has produced the same slice as DS-M_{tax} as shown in Figure 7.3(b) which includes statements S1', S2', S5' S7', S8', S9', S10', S11', S12' and S18'. The RCS produced in the second part of the Comparison Phase is used in this phase. The RCS includes statements S9', S10', S11' and S12'.

Test cases in RT, TC_4, TC_5 and TC_6, are sequently executed onto the UDS-M. Firstly, the TC_4 is executed onto UDS-M. The RTE_4 for TC_4 is S1', S2', S5', S7', S9', S10' and S18'. The RTE_4 contains only two members of RCS which

are statements S9' and S10'. This means that the RCS still does not achieve full coverage by the execution of the TC_4. TC_4 is included in the set of Optimised Regression Tests (RTO). Then, the TC_5 is executed onto UDS-M. The RTE_5 for TC_5 is similar to RTE_4 and contains only two members of RCS, statements S9' and S10'. Because the coverage of the RTE_5 onto RCS is similar to RTE_4, TC_5 is included in the RTO to replace its current member, TC_4. Next, the TC_6 is executed onto UDS-M. The RTE_6 for TC_6 is also similar to RTE_4 and contains only two members of RCS, statements S9' and S10'. Due to the fact that the coverage of the RTE_6 onto RCS is similar to RTE_5, then TC_6 is included in the RTO to replace its current member TC_5.

Although all the test cases in RT have been executed onto UDS-M, the RCS is still does not achieved full coverage. Only statements S9' and S10' from RCS are covered by these three test cases in RT. The remaining members of RCS, statements S11' and S12', are still not covered by any test cases. Therefore, the set RTO has only one test case which is TC_6. At the same time, the ReTSE model has flagged for additional new test cases because the coverage of RCS is still incomplete. A summary of the Optimisation Phase is given below:

$$\begin{aligned} \text{UDS-M} &= \{\text{S1', S2', S5' S7', S8', S9', S10', S11', S12', S18'}\} \\ \text{RCS} &= \{\text{S9', S10', S11', S12'}\} \end{aligned}$$

TC_4

RTE_4	=	{S1', S2', S5' S7', S9', S10', S18'}
RCS-current	=	{S9', S10'}
RCS-coverage	=	{S9', S10'}
RTO	=	$\{TC_4\}$
RCS-full	=	NO

TC_5

RTE_5	=	{S1', S2', S5' S7', S9', S10', S18'}
RCS-current	=	{S9', S10'}
RCS-coverage	=	{S9', S10'}
RTO	=	$\{TC_5\}$
RCS-full	=	NO

TC_6

RTE_6	=	{S1', S2', S5' S7', S9', S10', S18'}
RCS-current	=	{S9', S10'}
RCS-coverage	=	{S9', S10'}
RTO	=	$\{TC_6\}$
RCS-full	=	NO

The final output of the model for this case is given below:

RTO = $\{TC_6\}$

Request_New_Test_Cases = YES

7.2.3 Modification Type 3 - Delete Statements (Case 3)

In this case, The M has deleted a few statements in order to remove one condition for grading a tax code.

7.2.3.1 Phase 1: Program Analysis

Step 1.1: Pretty Print

The Pretty Print Step has produced a Certified Program (C) and Modified Program (M) as shown in Chapter 5, Figure 5.7 (page 76) and Figure 7.4 respectively.

Step 1.2: Slicing

In the Slicing Step, both C and M programs have four decomposition slices corresponding to four variables which are *income*, *tax*, *age* and *code*. The decomposition slices for C (DS-C) are shown in Figures 7.5- 7.8 in part (a). On the other hand, the decomposition slices for M (DS-M) are shown in part (b) in the same figures. The output summary of the Slicing Step is shown below:

- DS-C = {DS-C$_{income}$, DS-C$_{tax}$, DS-C$_{age}$, DS-C$_{code}$}
- DS-M = {DS-M$_{income}$, DS-M$_{tax}$, DS-M$_{age}$, DS-M$_{code}$}

7.2.3.2 Phase 2: Comparison

In the first part of the Comparison Phase, an output is produced from the *diff* tool as a result of the comparison between DS-C$_{age}$ and DS-M$_{age}$ (Figure 7.7). Therefore, both decomposition slices are included in a set of pairs of Difference

```
    #include <stdio.h>
    main()
    {
S1’   int income;
S2’   int tax;
S3’   int age;
S4’   char code;
S5’   scanf("%d", &income);
S6’   scanf("%d", &age);
S7’   if (income < 10000)
      {
S8’     tax = 0;
      }
      else
      {
S9’     income = income - 10000;
S10’    tax = (income*40/100);
      }
S11’  if (age < 65)
      {
S12’    code = ’L’;
      }
      else
      {
S13’    code = ’T’;
      }
S14’  printf("%d\n", tax);
S15’  printf("%c\n", code);
    }
```

Figure 7.4: Modified Program (M) (Case 3)

```
   #include <stdio.h>
   main()
   {
S1   int income;
S2   int tax;
S5   scanf("%d", &income);
S7   if (income < 10000)
     {
     }
     else
     {
S9     income=income-10000;
S10    tax=(income*40/100);
     }
   }
```

(a) DS-C$_{income}$

```
   #include <stdio.h>
   main()
   {
S1'  int income;
S2'  int tax;
S5'  scanf("%d", &income);
S7'  if (income < 10000)
     {
     }
     else
     {
S9'    income=income-10000;
S10'   tax=(income*40/100);
     }
   }
```

(b) DS-M$_{income}$

Figure 7.5: Decomposition Slice for Variable *income* (Case 3)

```
   #include <stdio.h>
   main()
   {
S1   int income;
S2   int tax;
S5   scanf("%d", &income);
S7   if (income < 10000)
     {
S8     tax = 0;
     }
     else
     {
S9     income=income-10000;
S10    tax=(income*40/100);
     }
S16  printf("%d\n", tax);
   }
```

(a) DS-C$_{tax}$

```
   #include <stdio.h>
   main()
   {
S1'  int income;
S2'  int tax;
S5'  scanf("%d", &income);
S7'  if (income < 10000)
     {
S8'    tax = 0;
     }
     else
     {
S9'    income=income-10000;
S10'   tax=(income*40/100);
     }
S14' printf("%d\n", tax);
   }
```

(b) DS-M$_{tax}$

Figure 7.6: Decomposition Slice for Variable *tax* (Case 3)

```
[L1]#include <stdio.h>
    main()
    {
S3    int age;
S6    scanf("%d", &age);
S11   if (age < 65)
      {
      }
S13   else if (age < 75)
[L10] {
      }
      else
      {
      }
    }
```

(a) DS-C$_{age}$

```
[L1]#include <stdio.h>
    main()
    {
S3'   int age;
S6'   scanf("%d", &age);
S11'  if (age < 65)
      {
[L8]  }
      else
[L10] {
      }
    }
```

(b) DS-M$_{age}$

Figure 7.7: Decomposition Slice for Variable *age* (Case 3)

```
[L1]#include <stdio.h>
    main()
    {
S3    int age;
S4    char code;
S6    scanf("%d", &age);
S11   if (age < 65)
      {
S12     code = 'L';
[L10] }
S13   else if (age < 75)
      {
S14     code = 'P';
      }
[L15] else
      {
S15     code = 'T';
      }
S17   printf("%c\n", code);
    }
```

(a) DS-C$_{code}$

```
[L1]#include <stdio.h>
    main()
    {
S3'    int age;
S4'    char code;
S6'    scanf("%d", &age);
S11'   if (age < 65)
       {
S12'     code = 'L';
[L10] }
       else
       {
S13'     code = 'T';
       }
S15'  printf("%c\n", code);
    }
```

(b) DS-M$_{code}$

Figure 7.8: Decomposition Slice for Variable *code* (Case 3)

Decomposition Slice (D). The DS-C$_{code}$ and DS-M$_{code}$ shown in Figure 7.8 are also included in the D because there is an output produced from the *diff* tool. The comparisons between DS-C$_{income}$ and DS-M$_{income}$ (Figure 7.5) and DS-C$_{tax}$ and DS-M$_{tax}$ (Figure 7.6) do not produce any output from the *diff* tool. Therefore, these decomposition slices are included in a set of pairs of Similar Decomposition Slice (S). No decomposition slice included in L and N. The output summary of the first part of the Comparison Phase is shown in Table 7.2.

Table 7.2: Comparison Results between DS-Cv_i and DS-Mv_i (Case 3)

Set of	Member of Set
D	{(DS-C$_{age}$, DS-M$_{age}$), (DS-C$_{code}$, DS-M$_{code}$)}
S	{(DS-C$_{income}$, DS-M$_{income}$), (DS-C$_{tax}$, DS-M$_{tax}$)}
L	{}
N	{}

Only the decomposition slices for variables age (DS-C$_{age}$ and DS-M$_{age}$) and code (DS-C$_{code}$ and DS-M$_{code}$) are involved in the second part of the Comparison Phase because they are members of the D. The comparison outputs using the *diff* tool are:

- Comparison between DS-C$_{age}$ and DS-M$_{age}$

```
9,11d8
<   else if (age < 75)
<   {
<   }
```

- Comparison between DS-C_{code} and DS-M_{code}

```
11,14d10
<   else if (age < 75)
<   {
<       code = 'P';
<   }
```

In the comparison between DS-C_{age} and DS-M_{age} (Figure 7.7), the statements in the range from line 9 ([L9]) to line 11 ([L11]) from DS-C_{age} are included in the set of Change Statements for variable *age* (CS_{age}). The statement at line 8 ([L8]) from DS-M_{age} is included in the set of Relevant Change Statements for variable *age* (RCS_{age}). In this case, line 8 in DS-M_{age} is not a statement, but a closed curly bracket (}), so that, the statement located immediately after line 8 is included in the RCS_{age}. Therefore, statement S13 from DS-C_{age} is included in the CS_{age} and no statement from DS-M_{age} is included in the RCS_{age}.

In the comparison between DS-C_{code} and DS-M_{code} (Figure 7.8), the statements in the range from line 11 ([L11]) to line 14 ([L14]) from DS-C_{code} are included in the set of Change Statements for variable *code* (CS_{code}). The statement at line 10 ([L10]) from DS-M_{code} is included in the set of Relevant Change Statements for variable *code* (RCS_{code}). In this case, line 10 in DS-M_{code} is not a statement but a closed curly bracket (}), so that, the statement located immediately after line 10 is included in the RCS_{code}. Therefore, statements S13 and S14 from DS-C_{code} are included in the CS_{code} and statement S13' from DS-M_{code} is included in the RCS_{code}. The CS is produced from the union of CS_{age} and CS_{code}. The RCS is produced from the union of RCS_{age} and RCS_{code}. A summary of the second part of the Comparison Phase is given below:

$$
\begin{aligned}
\text{CS} &= \text{CS}_{age} \cup \text{CS}_{code} \\
&= \{\text{S13}\} \cup \{\text{S3, S14}\} \\
&= \{\text{S13, S14}\}
\end{aligned}
$$

$$
\begin{aligned}
\text{RCS} &= \text{RCS}_{age} \cup \text{RCS}_{code} \\
&= \{\} \cup \{\text{S13'}\} \\
&= \{\text{S13'}\}
\end{aligned}
$$

7.2.3.3 Phase 3: Exclusion

The same test cases (TC) in Table 5.2 (page 84) and test histories (TH) in Table 5.3 (page 85) in Chapter 5 are used in this phase. Any TC_i where the CS is not a subset of TH_i will be included in the set of Excluded Test (ET). In this example, the statements S13 and S14, which are members of CS, are not subset of TH_1, TH_3, TH_4 and TH_6. Therefore, TC_1, TC_3, TC_4 and TC_6 will be included in the set ET. The remaining test cases in the Test Suite are included in the RT. Therefore, TC_2 and TC_5 are included in the RT. A summary of the Exclusion Phase is given below:

$$\text{RT} = \{\text{TC}_2, \text{TC}_5\}$$

7.2.3.4 Phase 4: Optimisation

In the Optimisation Phase, all TC_i that are members of RT will be executed onto UDS-M. In this example, there are only two DS-Mv_i members of D, DS-M_{age} and DS-M_{code}. The union of both decomposition slices (UDS-M) has produced the same slice as DS-M_{code} as shown in Figure 7.8(b) which includes statements S3', S4' S6', S11', S12', S13' and S15'. The RCS produced in the second part of the

Comparison Phase is used in this phase. The RCS only includes statement S13'.

There are only two test cases in the RT which are TC_2 and TC_5. Both test cases are executed onto the UDS-M. Firstly, the TC_2 is executed onto UDS-M. The RTE_2 for TC_2 is S3', S4', S6' S11', S13' and S15'. The RTE_2 contains a member of RCS which is statement S13'. This means that the RCS achieved full coverage by the execution of the TC_2. Then, the process will stop because the coverage of RCS is complete. Therefore, only TC_2 will be included in a set of RTO, while the TC_5 will be ignored. A summary of the Optimisation Phase is given below:

UDS-M = {S3', S4', S6' S11', S12', S13', S15'}

RCS = {S13'}

TC_2

RTE_2 = {S3', S4', S6' S11', S13', S15'}

RCS-current = {S13'}

RCS-coverage = {S13'}

RTO = $\{TC_2\}$

RCS-full = YES

The final output of the model for this case is given below:

RTO = $\{TC_2\}$

Request_New_Test_Cases = NO

7.2.4 Modification Type 4 - Add Variables (Case 4)

In this case, the M has two additional new variables which are *married* and *discount* in order to add new criteria to count the tax payment.

7.2.4.1 Phase 1: Program Analysis

Step 1.1: Pretty Print

The Pretty Print Step has produced a Certified Program (C) and Modified Program (M) as shown in Chapter 5, Figure 5.7 (page 76) and Figure 7.9 respectively.

Step 1.2: Slicing

In the Slicing Step, there are four decomposition slices for C that correspond to four variables which are *income*, *tax*, *age* and *code* as shown in Figures 7.10- 7.13 in part (a). Besides, there are six decomposition slices for M that correspond to six variables which are *income*, *tax*, *age*, *code*, *married* and *discount* (see Figures 7.10- 7.15 in part (b)). The M has two additional new variables (*married* and *discount*) compared to the C. The output summary of the Slicing Step is listed below:

- DS-C = {DS-C$_{income}$, DS-C$_{tax}$, DS-C$_{age}$, DS-C$_{code}$}
- DS-M = {DS-M$_{income}$, DS-M$_{tax}$, DS-M$_{age}$, DS-M$_{code}$, DS-M$_{married}$, DS-M$_{discount}$}

7.2.4.2 Phase 2: Comparison

In the first part of the Comparison Phase, the *diff* tool did not produce any output for the comparison between DS-C and DS-M for variables *income*, *tax*, *age*

```
    #include <stdio.h>
    main()
    {
S1’   int income;
S2’   int tax;
S3’   int age;
S4’   int married;
S5’   int discount;
S6’   char code;
S7’   scanf("%d", &income);
S8’   scanf("%d", &age);
S9’   scanf("%d", &married);
S10’  discount = 0;
S11’  if (income < 10000)
      {
S12’    tax = 0;
      }
      else
      {
S13’    income = income - 10000;
S14’    tax = (income*40/100);
      }
S15’  if (age < 65)
      {
S16’    code = ’L’;
      }
S17’  else if (age < 75)
      {
S18’    code = ’P’;
      }
      else
      {
S19’    code = ’T’;
      }
S20’  if (married)
      {
S21’    discount = 10;
      }
S22’  printf("%d\n", tax);
S23’  printf("%c\n", code);
S24’  printf("%c\n", discount);
    }
```

Figure 7.9: Modified Program (M) (Case 4)

```
    #include <stdio.h>
    main()
    {
S1    int income;
S2    int tax;
S5    scanf("%d", &income);
S7    if (income < 10000)
      {
      }
      else
      {
S9      income=income-10000;
S10     tax=(income*40/100);
      }
    }
```

(a) DS-C$_{income}$

```
    #include <stdio.h>
    main()
    {
S1'   int income;
S2'   int tax;
S7'   scanf("%d", &income);
S11'  if (income < 10000)
      {
      }
      else
      {
S13'    income=income-10000;
S14'    tax=(income*40/100);
      }
    }
```

(b) DS-M$_{income}$

Figure 7.10: Decomposition Slice for Variable *income* (Case 4)

```
    #include <stdio.h>
    main()
    {
S1    int income;
S2    int tax;
S5    scanf("%d", &income);
S7    if (income < 10000)
      {
S8      tax = 0;
      }
      else
      {
S9      income=income-10000;
S10     tax=(income*40/100);
      }
S16   printf("%d\n", tax);
    }
```

(a) DS-C$_{tax}$

```
    #include <stdio.h>
    main()
    {
S1'    int income;
S2'    int tax;
S7'    scanf("%d", &income);
S11'    if (income < 10000)
       {
S12'       tax = 0;
       }
       else
       {
S13'     income=income-10000;
S14'     tax=(income*40/100);
       }
S22'   printf("%d\n", tax);
    }
```

(b) DS-M$_{tax}$

Figure 7.11: Decomposition Slice for Variable *tax* (Case 4)

```
    #include <stdio.h>
    main()
    {
S3    int age;
S6    scanf("%d", &age);
S11   if (age < 65)
      {
      }
S13   else if (age < 75)
      {
      }
      else
      {
      }
    }
```

(a) DS-C$_{age}$

```
    #include <stdio.h>
    main()
    {
S3'   int age;
S8'   scanf("%d", &age);
S15'  if (age < 65)
      {
      }
S17'  else if (age < 75)
      {
      }
      else
      {
      }
    }
```

(b) DS-M$_{age}$

Figure 7.12: Decomposition Slice for Variable *age* (Case 4)

```
    #include <stdio.h>
    main()
    {
S3    int age;
S4    char code;
S6    scanf("%d", &age);
S11   if (age < 65)
      {
S12     code = 'L';
      }
S13   else if (age < 75)
      {
S14     code = 'P';
      }
      else
      {
S15     code = 'T';
      }
S17   printf("%c\n", code);
    }
```

(a) DS-C$_{code}$

```
    #include <stdio.h>
    main()
    {
S3'   int age;
S6'   char code;
S8'   scanf("%d", &age);
S15'  if (age < 65)
      {
S16'    code = 'L';
      }
S17'  else if (age < 75)
      {
S18'    code = 'P';
      }
      else
      {
S19'    code = 'T';
      }
S23'  printf("%c\n", code);
    }
```

(b) DS-M$_{code}$

Figure 7.13: Decomposition Slice for Variable *code* (Case 4)

```
(Not Exists)
```

(a) DS-C$_{married}$

```
    #include <stdio.h>
    main()
    {
S4'   char married;
S9'   scanf("%d", &married);
S20'  if (married)
      {
      }
    }
```

(b) DS-M$_{married}$

Figure 7.14: Decomposition Slice for Variable *married* (Case 4)

```
(Not Exists)
```

(a) DS-C$_{discount}$

```
    #include <stdio.h>
    main()
    {
S4'   int married;
S5'   char discount;
S9'   scanf("%d", &married);
S10'  discount = 0;
S20'  if (married)
      {
S21'    discount = 10;
      }
S24'  printf("%d\n", discount);
    }
```

(b) DS-M$_{discount}$

Figure 7.15: Decomposition Slice for Variable *discount* (Case 4)

and *code*. Therefore, these decomposition slices are included in the S. No decomposition slice included in D and L. However, DS-M$_{married}$ and DS-M$_{discount}$ are included in the set of New Decomposition Slices (N) because the decomposition slices for variables *married* and *discount* only exist in the DS-M, but not in the DS-C. The output summary of the first part of the Comparison Phase is shown in Table 7.3.

Table 7.3: Comparison Results between DS-Cv_i and DS-Mv_i (Case 4)

Set of	Member of Set
D	{}
S	{(DS-C$_{income}$, DS-M$_{income}$), (DS-C$_{tax}$, DS-M$_{tax}$) (DS-C$_{age}$, DS-M$_{age}$), (DS-C$_{code}$, DS-M$_{code}$)}
L	{}
N	{DS-M$_{married}$, DS-M$_{discount}$}

No comparison process is performed in the second part of the Comparison Phase because there is no decomposition slice in the D. Therefore, no statement is included in the set of CS and RCS. A summary of the second part of the Comparison Phase is given below:

$$\begin{aligned} \text{CS} &= \{\} \\ \text{RCS} &= \{\} \end{aligned}$$

7.2.4.3 Phase 3: Exclusion

The same TC in Table 5.2 (page 84) and TH in Table 5.3 (page 85) in Chapter 5 are used in this phase. Any TC$_i$ where the CS is not a subset of TH$_i$ will be included in the ET. If the CS is an empty set, then all test cases are included in

the ET and none in the RT. In this example, the CS is an empty set. Therefore, all test cases in the TS will be included in the ET and no test case is included in the RT. A summary of the Exclusion Phase is given below:

RT = {}

7.2.4.4 Phase 4: Optimisation

In the Optimisation Phase, all TC_i that are members of RT will be executed onto UDS-M. In this example, there is no test case in RT produced in the Exclusion Phase. However, the ReTSE model has flagged for additional new test cases because N is not empty. Therefore, the final output of the model for this case is given below:

RTO = {}

Request_New_Test_Cases = YES

7.2.5 Modification Type 5 - Delete Variables (Case 5)

In this case, the M has two deleted old variables which are *age* and *code* in order to remove a criterion to calculate the tax payment.

7.2.5.1 Phase 1: Program Analysis

Step 1.1: Pretty Print

The Pretty Print Step has produced a Certified Program (C) and Modified Program (M) as shown in Chapter 5, Figure 5.7 (page 76) and Figure 7.16 respectively.

```
     #include <stdio.h>
     main()
     {
S1'    int income;
S2'    int tax;
S3'    scanf("%d", &income);
S4'    if (income < 10000)
       {
S5'      tax = 0;
       }
       else
       {
S6'      income = income - 10000;
S7'      tax = (income*40/100);
       }
S8'    printf("%d\n", tax);
     }
```

Figure 7.16: Modified Program (M) (Case 5)

Step 1.2: Slicing

In the Slicing Step, there are four decomposition slices for C that correspond to four variables which are *income*, *tax*, *age* and *code* as shown in Figures 7.17-7.20 in part (a). Besides, there are only two decomposition slices for M that correspond to two variables which are *income* and *tax* as shown in Figure 7.17 and Figure 7.18 in part (b). The M has two deleted old variables compared to the C. The output summary of the Slicing Step is given below:

- DS-C = {DS-C$_{income}$, DS-C$_{tax}$, DS-C$_{age}$, DS-C$_{code}$}
- DS-M = {DS-M$_{income}$, DS-M$_{tax}$}

```
    #include <stdio.h>
    main()
    {
S1    int income;
S2    int tax;
S5    scanf("%d", &income);
S7    if (income < 10000)
      {
      }
      else
      {
S9      income=income-10000;
S10     tax=(income*40/100);
      }
    }
```

(a) DS-C$_{income}$

```
    #include <stdio.h>
    main()
    {
S1'   int income;
S2'   int tax;
S3'   scanf("%d", &income);
S4'   if (income < 10000)
      {
      }
      else
      {
S6'     income=income-10000;
S7'     tax=(income*40/100);
      }
    }
```

(b) DS-M$_{income}$

Figure 7.17: Decomposition Slice for Variable *income* (Case 5)

7.2.5.2 Phase 2: Comparison

In the first part of the Comparison Phase, the comparisons between DS-C and DS-M for variables *income* and *tax*, the *diff* tool did not produce any output.

```
    #include <stdio.h>
    main()
    {
S1    int income;
S2    int tax;
S5    scanf("%d", &income);
S7    if (income < 10000)
      {
S8      tax = 0;
      }
      else
      {
S9      income=income-10000;
S10     tax=(income*40/100);
      }
S16   printf("%d\n", tax);
    }
```

(a) DS-C$_{tax}$

```
    #include <stdio.h>
    main()
    {
S1'   int income;
S2'   int tax;
S3'   scanf("%d", &income);
S4'   if (income < 10000)
      {
S5'     tax = 0;
      }
      else
      {
S6'     income=income-10000;
S7'     tax=(income*40/100);
      }
S8'   printf("%d\n", tax);
    }
```

(b) DS-M$_{tax}$

Figure 7.18: Decomposition Slice for Variable *tax* (Case 5)

```
    #include <stdio.h>
    main()
    {
S3    int age;
S6    scanf("%d", &age);
S11   if (age < 65)
      {
      }
S13   else if (age < 75)
      {
      }
      else
      {
      }
    }
```

(a) DS-C$_{age}$

(Not Exists)

(b) DS-M$_{age}$

Figure 7.19: Decomposition Slice for Variable *age* (Case 5)

```
    #include <stdio.h>
    main()
    {
S3    int age;
S4    char code;
S6    scanf("%d", &age);
S11   if (age < 65)
      {
S12     code = 'L';
      }
S13   else if (age < 75)
      {
S14     code = 'P';
      }
      else
      {
S15     code = 'T';
      }
S17   printf("%c\n", code);
    }
```

(a) DS-C$_{code}$

```
(Not Exists)
```

(b) DS-M$_{code}$

Figure 7.20: Decomposition Slice for Variable *code* (Case 5)

Therefore, these decomposition slices are included in the S. No decomposition slice is included in D and N. However, DS-C$_{age}$ and DS-C$_{code}$ are included in L because the decomposition slices for *age* and *code* only exist in the DS-C, but not in the DS-M. The output summary of the first part of the Comparison Phase is shown in Table 7.4.

Table 7.4: Comparison Results between DS-Cv_i and DS-Mv_i (Case 5)

Set of	Member of Set
D	{}
S	{(DS-C$_{income}$, DS-M$_{income}$), (DS-C$_{tax}$, DS-M$_{tax}$)}
L	{DS-C$_{age}$, DS-C$_{code}$}
N	{}

No comparison process is performed at the second part of the Comparison Phase because there is no decomposition slice in D. Therefore, there is no statement is included in the set of CS and RCS. A summary of the second part of the Comparison Phase is given below:

$$\begin{aligned} \text{CS} &= \{\} \\ \text{RCS} &= \{\} \end{aligned}$$

7.2.5.3 Phase 3: Exclusion

The same TC in Table 5.2 (page 84) and TH in Table 5.3 (page 85) as shown in Chapter 5 are used in this phase. Any TC_i where the CS is not a subset of TH_i will be included in the set of Excluded Test (ET). If the CS is an empty set, then all the test cases are included in the ET and none in the RT. In this example, the CS is an empty set. Therefore, all the test cases in the TS will be included in the set of ET while none is included in the set of Regression Tests (RT). The Exclusion Phase can be summarised as below:

$$\text{RT} = \{\}$$

7.2.5.4 Phase 4: Optimisation

In the Optimisation Phase, all TC_i that are members of RT will be executed onto UDS-M. In this case, there is no test case produced for RT in the Exclusion Phase. Therefore, the final output of the model for this case is given below:

$$\text{RTO} = \{\}$$

$$\text{Request\_New\_Test\_Cases} = \text{NO}$$

7.3 Combination of Modification Types

7.3.1 Combination 1: Change, Add and Delete Statements (Case 6)

In this case, the M has two additional new statements at S9' and S10' (in Modified Program), change at statement S12' (in Modified Program) and delete statements at S13 and S14 (in Certified Program) in order to change the condition for tax calculation.

7.3.1.1 Phase 1: Program Analysis

Step 1.1: Pretty Print

The Pretty Print Step has produced a Certified Program (C) and Modified Program (M) (refer to Figure 5.7 in Chapter 5 (page 76) and Figure 7.21).

Step 1.2: Slicing

In the Slicing Step, both C and M programs have four decomposition slices correspond to variables *income*, *tax*, *age* and *code*. The decomposition slices for C are DS-C$_{income}$, DS-C$_{tax}$, DS-C$_{age}$ and DS-C$_{code}$ as shown in part (a) of Figures 7.22- 7.25. The decomposition slices for M are DS-M$_{income}$, DS-M$_{tax}$, DS-M$_{age}$ and DS-M$_{code}$ as shown in part (b) in the same figures. The output summary of the Slicing Step is given below:

- DS-C = {DS-C$_{income}$, DS-C$_{tax}$, DS-C$_{age}$, DS-C$_{code}$}
- DS-M = {DS-M$_{income}$, DS-M$_{tax}$, DS-M$_{age}$, DS-M$_{code}$}

```
    #include <stdio.h>
    main()
    {
S1’   int income;
S2’   int tax;
S3’   int age;
S4’   char code;
S5’   scanf("%d", &income);
S6’   scanf("%d", &age);
S7’   if (income < 10000)
      {
S8’     tax = 0;
      }
S9’   else if (income < 20000)
      {
S10’    tax=((income-10000)*25/100);
      }
      else
      {
S11’    income=income-10000;
S12’    tax=(income*30/100);
      }
S13’  if (age < 65)
      {
S14’    code = ’L’;
      }
      else
      {
S15’    code = ’T’;
      }
S16’  printf("%d\n", tax);
S17’  printf("%c\n", code);
    }
```

Figure 7.21: Modified Program (M) (Case 6)

```
[L1]#include <stdio.h>
    main()
    {
S1    int income;
S2    int tax;
S5    scanf("%d", &income);
S7    if (income < 10000)
      {
      }
[L10] else
      {
S9      income=income-10000;
S10     tax=(income*40/100);
      }
    }
```

(a) DS-C$_{income}$

```
[L1]#include <stdio.h>
    main()
    {
S1'  int income;
S2'  int tax;
S5'  scanf("%d", &income);
S7'  if (income < 10000)
     {
[9]  }
S9'  else if (income < 20000)
     {
S10'  tax=((income-10000)*25/100);
     }
     else
[L15]{
S11'  income=income-10000;
S12'  tax=(income*30/100);
     }
    }
```

(b) DS-M$_{income}$

Figure 7.22: Decomposition Slice for Variable *income* (Case 6)

```
[L1]#include <stdio.h>
    main()
    {
S1    int income;
S2    int tax;
S5    scanf("%d", &income);
S7    if (income < 10000)
      {
S8      tax = 0;
[L10] }
      else
      {
S9      income=income-10000;
S10     tax=(income*40/100);
      }
S16   printf("%d\n", tax);
    }
```

(a) DS-C$_{tax}$

```
[L1]#include <stdio.h>
    main()
    {
S1'   int income;
S2'   int tax;
S5'   scanf("%d", &income);
S7'   if (income < 10000)
      {
S8'     tax = 0;
[L10] }
S9'   else if (income < 20000)
      {
S10'    tax=((income-10000)*25/100);
      }
[L15] else
      {
S11'    income=income-10000;
S12'    tax=(income*30/100);
      }
S16'  printf("%d\n", tax);
    }
```

(b) DS-M$_{tax}$

Figure 7.23: Decomposition Slice for Variable *tax* (Case 6)

```
[L1]#include <stdio.h>
    main()
    {
S3    int age;
S6    scanf("%d", &age);
S11   if (age < 65)
      {
      }
S13   else if (age < 75)
[L10] {
      }
      else
      {
      }
    }
```

(a) DS-C$_{age}$

```
[L1]#include <stdio.h>
    main()
    {
S3'   int age;
S6'   scanf("%d", &age);
S13'  if (age < 65)
      {
      }
[L10]   else
      {
      }
    }
```

(b) DS-M$_{age}$

Figure 7.24: Decomposition Slice for Variable *age* (Case 6)

```
[L1]#include <stdio.h>
    main()
    {
S3    int age;
S4    char code;
S6    scanf("%d", &age);
S11   if (age < 65)
      {
S12     code = 'L';
[L10] }
S13   else if (age < 75)
      {
S14     code = 'P';
      }
[L15] else
      {
S15     code = 'T';
      }
S17   printf("%c\n", code);
    }
```

(a) DS-C$_{code}$

```
[L1]#include <stdio.h>
    main()
    {
S3'   int age;
S4'   char code;
S6'   scanf("%d", &age);
S13'  if (age < 65)
      {
S14'    code = 'L';
[L10] }
      else
      {
S15'    code = 'T';
      }
S17'  printf("%c\n", code);
    }
```

(b) DS-M$_{code}$

Figure 7.25: Decomposition Slice for Variable *code* (Case 6)

7.3.1.2 Phase 2: Comparison

In the first part of the Comparison Phase, the comparisons between DS-C and DS-M for variables *income* (DS-C$_{income}$ and DS-M$_{income}$), *tax* (DS-C$_{tax}$ and DS-M$_{tax}$), *age* (DS-C$_{age}$ and DS-M$_{age}$) and *code* (DS-C$_{code}$ and DS-M$_{code}$) have produced an output from the *diff* tool. Therefore, these decomposition slices are included in the set of pairs of Difference Decomposition Slice (D). No decomposition slice is included in the S, L and N. The output summary of the first part of the Comparison Phase is shown in Table 7.5.

Table 7.5: Comparison Results between DS-Cv_i and DS-Mv_i (Case 6)

Set of	Member of Set
D	{(DS-C$_{income}$, DS-M$_{income}$), (DS-C$_{tax}$, DS-M$_{tax}$) (DS-C$_{age}$, DS-M$_{age}$), (DS-C$_{code}$, DS-M$_{code}$)}
S	{}
L	{}
N	{}

All the decomposition slices are involved in the second part of the Comparison Phase because they are members of D. The comparison output using the *diff* tool is illustrated below:

- Comparison between DS-C$_{income}$ and DS-M$_{income}$

```
9a10,13
>   else if (income < 20000)
>  {
>     tax = ((income-10000)*25/100);
>  }

13c17
<  tax = (income*40/100);
- - -
>  tax = (income*30/100);
```

- Comparison between DS-C$_{tax}$ and DS-M$_{tax}$

```
10a11,14
>  else if (income < 20000)
>  {
>    tax = ((income-10000)*25/100);
>  }

14c18
<  tax = (income*40/100);
- - -
>  tax = (income*30/100);
```

- Comparison between DS-C$_{age}$ and DS-M$_{age}$

```
9,11d8
<  else if (age < 75)
<  {
<  }
```

- Comparison between DS-C$_{code}$ and DS-M$_{code}$

```
11,14d10
<  else if (age < 75)
<  {
<     code = 'P';
<  }
```

In the comparison between DS-C_{income} and DS-M_{income} (Figure 7.22), the statements at lines 9 ([L9]) and 13 ([L13]) from DS-C_{income} are included in the set of Change Statements for variable *income* (CS_{income}). The statements in the range from line 10 ([L10]) to line 13 ([L13]) and at line 17 ([L17]) from DS-M_{income} are included in the set of Relevant Change Statements for variable *income* (RCS_{income}). Line 9 is not a statement but a closed curly bracket (}), so that, the statement located immediately after line 9 is included in the CS_{income}. Therefore, statements S9 and S10 from DS-C_{income} are included in the CS_{income} while statements S9', S10' and S12' from DS-M_{income} are included in the RCS_{income}. Statement S11' is also included in the RCS_{income} because it located at the same branch of statement S12'.

In the comparison between DS-C_{tax} and DS-M_{tax} (Figure 7.23), the statements at lines 10 ([L10]) and 14 ([L14]) from DS-C_{tax} are included in the set of Change Statements for variable *tax* (CS_{tax}). The statements in the range from line 11 ([L11]) to line 14 ([L14]) and at line 18 ([L18]) from DS-M_{tax} are included in the set of Relevant Change Statements for variable *tax* (RCS_{tax}). Line 10 is not a statement but a closed curly bracket (}), so that, the statement located immediately after line 10 is included in the CS_{tax}. Therefore, statements S9 and S10 from DS-C_{tax} are included in the CS_{tax} and statements S9', S10' and S12' from DS-M_{tax} are included in the RCS_{tax}. Statement S11' is also included in the RCS_{tax} because it located at the same branch of statement S12'.

In the comparison between DS-C_{age} and DS-M_{age} (Figure 7.24), the statement in the range from line 9 ([L9]) to line 11 ([L11]) from DS-C_{age} is included in the

set of Change Statements for variable *age* (CS_{age}). The statement at line 8 ([L8]) from DS-M_{age} is included in the set of Relevant Change Statements for variable *age* (RCS_{age}). In this case, line 8 in DS-M_{age} is not a statement but a closed curly bracket (}), so that, the statement located immediately after line 8 is included in the RCS_{age}. Therefore, statement S13 from DS-C_{age} is included in the CS_{age} while none from DS-M_{age} is included in the RCS_{age}.

In the comparison between DS-C_{code} and DS-M_{code} (Figure 7.25), the statement in the range from line 11 ([L11]) to line 14 ([L14]) from DS-C_{code} is included in the set of Change Statements for variable *code* (CS_{code}). The statement at line 10 ([L10]) from DS-M_{code} is included in the set of Relevant Change Statements for variable *code* (RCS_{code}). In this case, line 10 in DS-M_{code} is not a statement but a closed curly bracket (}), so that, the statement located immediately after line 10 is included in the RCS_{code}. Therefore, statements S13 and S14 from DS-C_{code} are included in the CS_{code} and statement S15' from DS-M_{code} is included in the RCS_{code}. The CS is produced from the union of CS_{tax}, CS_{income}, CS_{age} and CS_{code} while the RCS is produced from the union of RCS_{tax}, RCS_{income}, RCS_{age} and RCS_{code}. A summary of the second part of the Comparison Phase is given below:

$$\begin{aligned} CS &= CS_{income} \cup CS_{tax} \cup CS_{age} \cup CS_{code} \\ &= \{S9, S10\} \cup \{S9, S10\} \cup \{S13\} \cup \{S13, S14\} \\ &= \{S9, S10, S13, S14\} \end{aligned}$$

$$\begin{aligned} \text{RCS} &= \text{RCS}_{income} \cup \text{RCS}_{tax} \cup \text{RCS}_{age} \cup \text{RCS}_{code} \\ &= \{\text{S9', S10', S11', S12'}\} \cup \{\text{S9', S10', S11', S12'}\} \cup \{\} \cup \{\text{S15'}\} \\ &= \{\text{S9', S10', S11', S12', S15'}\} \end{aligned}$$

7.3.1.3 Phase 3: Exclusion

The same TC in Table 5.2 (page 84) and TH in Table 5.3 (page 85) as shown in Chapter 5 are used in this phase. Any TC_i where the CS is not a subset of TH_i will be included in the ET. In this example, the CS is not subset of TH_1, TH_2, TH_3, TH_4 and TH_6. Therefore, TC_1, TC_2, TC_3, TC_4, and TC_6 will be included in the ET. The remaining test cases in the Test Suite are included in the RT. Therefore, only TC_5 is included in the RT. A summary of the Exclusion Phase is given below:

$$\text{RT} = \{\text{TC}_5\}$$

7.3.1.4 Phase 4: Optimisation

In the Optimisation Phase, there are four DS-Mv_i members of D which are DS-M_{income}, DS-M_{tax}, DS-M_{age} and DS-M_{code}. The union of these decomposition slices (UDS-M) has produced the same program as the M as shown in Figure 7.21 which includes statements S1', S2', S3', S4', S5', S6', S7', S8', S9', S10', S11', S12', S13', S14', S15', S16' and S17'. The RCS produced in the second part of the Comparison Phase is used in this phase. The RCS includes statements S9', S10', S11', S12' and S15'.

The TC_5 from RT is executed onto the UDS-M. The RTE_5 for TC_5 is S1', S2', S3', S4', S5', S6', S7', S9', S10', S13', S15', S16' and S17'. The RTE_5 only

contains three members of RCS, statements S9', S10' and S15'. This means that the the coverage of RCS is incomplete by the execution of the TC_5. Therefore, the $TC_5$is included in the RTO. At the same time, the model has flagged for additional new test cases. A summary of the Optimisation Phase is given below:

UDS-M = {S1', S2', S3', S4', S5', S6', S7', S8', S9', S10',
S11', S12', S13', S14', S15', S16', S17'}

RCS = {S9', S10', S11', S12', S15'}

TC_5

RTE_5 = {S1',S2',S3',S4',S5',S6',S7',S9',S10',S13',S15',S16',S17'}

RCS-current = {S9', S10', S15'}

RCS-coverage = {S9', S10', S15'}

RTO = $\{TC_5\}$

RCS-full = NO

The final output of the model for this case is given below:

RTO = $\{TC_5\}$

Request_New_Test_Cases = YES

7.3.2 Combination 2: All Types of Modification (Case 7)

In this case, the M has included all types of modifications except the delete statements.

7.3.2.1 Phase 1: Program Analysis

Step 1.1: Pretty Print

The Pretty Print Step has produced a Certified Program (C) and Modified Program (M) (refer to Figure 5.7 in Chapter 5 (page 76) and Figure 7.26).

Step 1.2: Slicing

In the Slicing Step, both C and M programs have four decomposition slices. The decomposition slices for C are DS-C$_{income}$, DS-C$_{tax}$, DS-C$_{age}$ and DS-C$_{code}$ that correspond to variables *income*, *tax*, *age* and *code* as shown in part (a) of Figures 7.27- 7.30. The decomposition slices for M are DS-M$_{income}$, DS-M$_{tax}$, DS-M$_{married}$ and DS-M$_{discount}$ that correspond to variables *income*, *tax*, *married* and *discount* as shown in part (b) of Figure 7.27, Figure 7.28, Figure 7.31 and Figure 7.32 respectively. The output summary of the Slicing Step is given below:

- DS-C = {DS-C$_{income}$, DS-C$_{tax}$, DS-C$_{age}$, DS-C$_{code}$}
- DS-M = {DS-M$_{income}$, DS-M$_{tax}$, DS-M$_{married}$, DS-M$_{discount}$}

7.3.2.2 Phase 2: Comparison

In the first part of the Comparison Phase, the comparisons between DS-C and DS-M for variables *income* (DS-C$_{income}$ and DS-M$_{income}$) and *tax* (DS-C$_{tax}$ and

```
    #include <stdio.h>
    main()
    {
S1’   int income;
S2’   int tax;
S3’   int married;
S4’   int discount;
S5’   scanf("%d", &income);
S6’   scanf("%d", &married);
S7’   discount = 0;
S8’   if (income < 10000)
      {
S9’     tax = 0;
      }
S10’    else if (income < 20000)
      {
S11’    tax = ((income-10000)*25/100);
      }
      else
      {
S12’    income = income - 10000;
S13’    tax = (income*30/100);
      }
S14’  if (married)
      {
S15’    discount = 10;
      }
S16’  printf("%d\n", tax);
S17’  printf("%c\n", discount);
    }
```

Figure 7.26: Modified Program (M) (Case 7)

```
[L1]#include <stdio.h>
    main()
    {
S1    int income;
S2    int tax;
S5    scanf("%d", &income);
S7    if (income < 10000)
      {
      }
[L10] else
      {
S9      income=income-10000;
S10     tax=(income*40/100);
      }
    }
```

(a) DS-C$_{income}$

```
[L1]#include <stdio.h>
    main()
    {
S1'   int income;
S2'   int tax;
S5'   scanf("%d", &income);
S7'   if (income < 10000)
      {
[L9]  }
S9'   else if (income < 20000)
      {
S10'    tax=((income-10000)*25/100);
      }
      else
      {
S11'    income=income-10000;
S12'    tax=(income*30/100);
      }
    }
```

(b) DS-M$_{income}$

Figure 7.27: Decomposition Slice for Variable *income* (Case 7)

```
[L1]#include <stdio.h>
    main()
    {
S1    int income;
S2    int tax;
S5    scanf("%d", &income);
S7    if (income < 10000)
      {
S8      tax = 0;
[L10] }
      else
      {
S9      income=income-10000;
S10     tax=(income*40/100);
      }
S16   printf("%d\n", tax);
    }
```

(a) DS-C$_{tax}$

```
[L1]#include <stdio.h>
    main()
    {
S1'   int income;
S2'   int tax;
S5'   scanf("%d", &income);
S7'   if (income < 10000)
      {
S8'     tax = 0;
[L10] }
S9'   else if (income < 20000)
      {
S10'    tax=((income-10000)*25/100);
      }
[L15] else
      {
S11'    income=income-10000;
S12'    tax=(income*30/100);
      }
S16'  printf("%d\n", tax);
    }
```

(b) DS-M$_{tax}$

Figure 7.28: Decomposition Slice for Variable *tax* (Case 7)

```
    #include <stdio.h>
    main()
    {
S3    int age;
S6    scanf("%d", &age);
S11   if (age < 65)
      {
      }
S13   else if (age < 75)
      {
      }
      else
      {
      }
    }
```

(a) DS-C$_{age}$

```
(Not Exists)
```

(b) DS-M$_{age}$

Figure 7.29: Decomposition Slice for Variable *age* (Case 7)

```
    #include <stdio.h>
    main()
    {
S3    int age;
S4    char code;
S6    scanf("%d", &age);
S11   if (age < 65)
      {
S12     code = 'L';
      }
S13   else if (age < 75)
      {
S14     code = 'P';
      }
      else
      {
S15     code = 'T';
      }
S17   printf("%c\n", code);
    }
```

(a) DS-C$_{code}$

```
(Not Exists)
```

(b) DS-M$_{code}$

Figure 7.30: Decomposition Slice for Variable *code* (Case 7)

```
(Not Exists)
```

(a) DS-C$_{married}$

```
    #include <stdio.h>
    main()
    {
S4'   char married;
S9'   scanf("%d", &married);
S20'  if (married)
      {
      }
    }
```

(b) DS-M$_{married}$

Figure 7.31: Decomposition Slice for Variable *married* (Case 7)

```
(Not Exists)
```

(a) DS-C$_{discount}$

```
    #include <stdio.h>
    main()
    {
S4'   int married;
S5'   char discount;
S9'   scanf("%d", &married);
S10'  discount = 0;
S20'  if (married)
      {
S21'    discount = 10;
      }
S24'  printf("%d\n", discount);
    }
```

(b) DS-M$_{discount}$

Figure 7.32: Decomposition Slice for Variable *discount* (Case 7)

DS-M$_{tax}$) have produced an output from the *diff* tool. Therefore, these decomposition slices are included in the D. No decomposition slice is included in the S. However, the DS-C$_{age}$ (Figure 7.29) and DS-C$_{code}$ (Figure 7.30) are included in the L because the decomposition slices for variables *age* and *code* only exist in the DS-C, but not in the DS-M. The DS-M$_{married}$ (Figure 7.31) and DS-M$_{discount}$ (Figure 7.32) are included in the N because the decomposition slices for variable *married* and *discount* only exist in the DS-M and not the DS-C. The output summary of the first part of the Comparison Phase is summarised in Table 7.6.

Table 7.6: Comparison Results between DS-Cv_i and DS-Mv_i (Case 7)

Set of	Member of Set
D	{(DS-C$_{income}$, DS-M$_{income}$), (DS-C$_{tax}$, DS-M$_{tax}$)}
S	{}
L	{DS-C$_{age}$, DS-C$_{code}$}
N	{DS-C$_{married}$, DS-C$_{discount}$}

In this case, only the decomposition slices for variables *tax* (DS-C$_{tax}$ and DS-M$_{tax}$) and *income* (DS-C$_{income}$ and DS-M$_{income}$) are involved in the second part of the Comparison Phase because they are members of D. The comparison output using the *diff* tool is given below:

- Comparison between DS-C$_{income}$ and DS-M$_{income}$

```
9a10,13
>   else if (income < 20000)
>   {
>      tax = ((income-10000)*25/100);
>   }

13c17
<   tax = (income*40/100);
- - -
>   tax = (income*30/100);
```

- Comparison between DS-C$_{tax}$ and DS-M$_{tax}$

```
10a11,14
>   else if (income < 20000)
>   {
>      tax = ((income-10000)*25/100);
>   }

14c18
<   tax = (income*40/100);
- - -
>   tax = (income*30/100);
```

In the comparison between DS-C$_{income}$ and DS-M$_{income}$ (Figure 7.27), the statements at lines 9 ([L9]) and 13 ([L13]) from DS-C$_{income}$ are included in the CS$_{income}$. The statements in the range from line 10 ([L10]) to line 13 ([L13]) and at line 17 ([L17]) from DS-M$_{income}$ are included in the RCS$_{income}$. Line 9 is not a statement but a closed curly bracket (}), so that, the statement located immediately after line 9 is included in the CS$_{income}$. Therefore, statements S9 and S10 from DS-C$_{income}$ are included in the CS$_{income}$ while statements S9', S10' and S12' from DS-M$_{income}$ are included in the RCS$_{income}$. Statement S11' is also included in the RCS$_{income}$ because it is located at the same branch of statement S12'.

In the comparison between DS-C_{tax} and DS-M_{tax} (Figure 7.28), the statements at lines 10 ([L10]) and 14 ([L14]) from DS-C_{tax} are included in the CS_{tax}. The statements in the range from line 11 ([L11]) to line 14 ([L14]) and at line 18 ([L18]) from DS-M_{tax} are included in the RCS_{tax}. Line 10 is not a statement but a closed curly bracket (}), so that, the statement located immediately after line 10 is included in the CS_{tax}. Therefore, statements S9 and S10 from DS-C_{tax} are included in the CS_{tax} while statements S9', S10' and S12' from DS-M_{tax} are included in the RCS_{tax}. Statement S11' is also included in the RCS_{tax} because it is located at the same branch of statement S12'. The CS is produced from the union of CS_{tax} and CS_{income} while the RCS is produced from the union of RCS_{tax} and RCS_{income}. A summary of the second part of the Comparison Phase is given below:

$$\begin{aligned} CS &= CS_{income} \cup CS_{tax} \\ &= \{S9, S10\} \cup \{S9, S10\} \\ &= \{S9, S10\} \end{aligned}$$

$$\begin{aligned} RCS &= RCS_{income} \cup RCS_{tax} \\ &= \{S9', S10', S11', S12'\} \cup \{S9', S10', S11', S12'\} \\ &= \{S9', S10', S11', S12'\} \end{aligned}$$

7.3.2.3 Phase 3: Exclusion

The same TC in Table 5.2 (page 84) and TH in Table 5.3 (page 85) as shown in Chapter 5 are used in this phase. Any TC_i where the CS is not a subset of TH_i will be included in the ET. In this example, the CS is not subset of TH_1, TH_2 and

TH_3. Therefore, TC_1, TC_2 and TC_3 will be included in the ET. The remaining test cases in the Test Suite are included in the RT. Therefore, TC_4, TC_5 and TC_6 are included in the RT. A summary of the Exclusion Phase is given below:

$$RTO = RT = \{TC_4, TC_5, TC_6\}$$

7.3.2.4 Phase 4: Optimisation

In the Optimisation Phase, all TC_i that are members of RT will be executed onto UDS-M. In this example, there are two DS-Mv_i members of D, DS-M_{income} and DS-M_{tax}. The union of these decomposition slices (UDS-M) has produced the same slice as DS-M_{tax} as shown in Figure 7.28(b) which includes statements S1', S2', S5', S7', S8', S9', S10', S11', S12' and S16'. The RCS produced in the second part of the Comparison Phase is used in this phase. The RCS includes statements S9', S10', S11' and S12'.

Test cases in RT, TC_4, TC_5 and TC_6 are sequently executed onto the UDS-M. Firstly, the TC_4 is executed onto UDS-M. The RTE_4 for TC_4 is S1', S2', S5', S7', S9', S10' and S16'. The RTE_4 contains only two members of RCS which are statements S9' and S10'. This means that the coverage of RCS is still incomplete by the execution of the TC_4. TC_4 is included in the RTO. Next, the TC_5 is executed onto UDS-M. The RTE_5 for TC_5 is similar to RTE_4 and contains only two members of RCS which are statements S9' and S10'. Due to the coverage of the RTE_5 onto RCS is similar to RTE_4, then the TC_5 is included in the RTO to replace its current member, TC_4. After that, the TC_6 is executed onto UDS-M. The RTE_6 for TC_6 is also similar to RTE_5. Due to the coverage of the RTE_6 onto

RCS is similar to RTE_5, then TC_6 is included in the RTO to replace its current member, TC_5.

Although all test cases in RT have been executed onto the UDS-M, the coverage of RCS is still incomplete. Only statements S9' and S10' from RCS are covered by these three test cases in RT. The remaining statements S11' and S12' in the RCS are still not covered by any test cases. Therefore, the RTO has only one test case which is TC_6. At the same time, the model has flagged for additional new test cases because the coverage of RCS is still incomplete. A summary of the Optimisation Phase is given below:

UDS-M = {S1',S2',S5',S7',S8',S9',S10',S11',S12',S16'}

RCS = {S9',S10',S11',S12'}

TC_4

RTE_4 = {S1', S2', S5', S7', S9', S10', S16'}

RCS-current = {S9', S10'}

RCS-coverage = {S9', S10'}

RTO = $\{TC_4\}$

RCS-full = NO

TC_5

RTE_5 = {S1', S2', S5', S7', S9', S10', S16'}

RCS-current = {S9', S10'}

RCS-coverage = {S9', S10'}

RTO = $\{TC_5\}$

RCS-full = NO

TC_6

RTE_6	=	{S1', S2', S5', S7', S9', S10', S16'}
RCS-current	=	{S9', S10'}
RCS-coverage	=	{S9', S10'}
RTO	=	$\{TC_6\}$
RCS-full	=	NO

The final output of the model for this case is given below:

RTO = $\{TC_6\}$

Request_New_Test_Cases = YES

7.4 Summary

The chapter has discussed seven different case studies. The chapter starts with the five case studies that represent the five types of modification. Each of these case studies focuses on one type of modification at a time. Then, another two case studies are presented for multiple modification types at a time. The ReTSE model works for all of these case studies. Further analysis of the results of these case studies is presented in the next chapter.

Chapter 8

Analysis and Evaluation

8.1 Introduction

This chapter presents an analysis and evaluation of the ReTSE model. The analysis refers to the case studies discussed in Chapter 7. This analysis is presented in the second section. The evaluation is divided into two parts. The first part is an evaluation based on an existing evaluation framework proposed by Rothermel and Harrold [88] as described in Chapter 3, Section 3.2 (page 20). The second part is an evaluation of the comparison between the ReTSE model and the Pythia technique proposed by Vokolos and Frankl [102] which is described in Chapter 3, Section 3.5.2 (page 25). The first part is presented in the third section and the second part is presented in the forth section.

8.2 Analysis of Case Studies

Table 8.1 shows the summary of all case studies that have been presented in Chapter 7. There are three significant results. First, the number of slices in the set of pairs of Difference Decomposition Slices (D) is the main factor to produce the number of test cases selected for a set of Regression Tests (RT) and a set of Optimised Regression Tests (RTO). Test cases are only selected for RT and RTO if and only if D is not empty. This can be seen in Case 4 (page 115) and Case 5 (page 122) where there are no test cases selected for RT and RTO because D is empty. This means that by using this model, any instances of add or delete variables in the Modified Program (M) that do not have slices in D will not select any test cases for RT and RTO. In the other cases (Case 1, Case 2, Case 3, Case 6 and Case 7), the model has produced at least one test case for RT and RTO due to D having some slices. This shows the significance of using the decomposition slicing technique in the model. The decomposition slicing technique is capable of detecting the situation of add and delete variables in M that can ignore the need of selecting test cases.

The second significant result is the reduction of the number of test cases for RT and RTO from the existing Test Suite (TS). There are six test cases in TS that have been used in all the case studies. The reduction is divided into two parts. The first part is after the Exclusion Phase which produces the RT. The second part is after the Optimisation Phase which produces the RTO. At the Exclusion Phase, the ReTSE model has reduced more than 50% of test cases in TS for RT. At some point, especially in Case 4 and Case 5, the model has reduced 100% of

Table 8.1: Summary of Case Studies

	Case 1 Change Statements	Case 2 Add Statements	Case 3 Delete Statements	Case 4 Add New Variables	Case 5 Delete Variables	Case 6 Change, Add & Delete Statement	Case 7 All Types of Modification (except Delete Statements)
No. of Statements in C	17	17	17	17	17	17	17
No. of Statements in M	17	19	15	24	8	17	17
No. of slices in DS-C	4	4	4	4	4	4	4
No. of slices in DS-M	4	4	4	6	2	4	4
No. of pair slices in D	2	2	2	0	0	4	2
No. of pair slices in S	2	2	2	4	2	0	0
No. of slices in L	0	0	0	0	2	0	2
No. of slices in N	0	0	0	2	0	0	2
No. of statements in CS	1	1	2	0	0	4	2
No. of statements in RCS	2	4	1	0	0	5	4
No. of test cases in TS	6	6	6	6	6	6	6
No. of test cases in RT	3	3	2	0	0	1	3
Test Cases Reduction % ((TS-RT)/TS x 100)	50	50	66.67	100	100	83.3	50
No. of test cases in RTO	1	1	1	0	0	1	1
Test Cases Reduction % ((RT-RTO)/RT x 100)	66.67	66.67	50	-	-	-	66.67
Decision for RNTC (Request New Test Cases)	NO	YES	NO	YES	NO	YES	YES

test cases in TS for RT. That means no test case is selected for RT. In the Optimisation Phase, the model once again has reduced at least 50% of test cases from RT for RTO as shown in Case 1, Case 2, Case, 3 and Case 7. The other cases are not relevant because RT has only one test case or none. These results show that the ReTSE model is capable to give a significant reduction of test cases to test the Modified Program (M). However, these results are only based on case studies that used small set of test cases. In future, the case studies should used large set of test cases in order to get concise results. This can be proceed when the model scale-up with inter-procedural concept in order to tackle large programs.

The third significant result is the ability of the ReTSE model to decide whether a new test case is needed. This can be seen at the bottom of Table 8.1. The model has determined that there are no new test cases needed in Case 1, Case 3 and Case 5, but Case 2, Case 4, Case 6 and Case 7 require new test cases for the M. However, the process of designing additional new test cases is beyond the scope of the ReTSE model.

8.3 Evaluation of the ReTSE Model

Rothermel and Harrold's framework for evaluating selective regression testing techniques consists of four categories which are inclusiveness, precision, efficiency and generality [88]. The framework has been described in Chapter 3, Section 3.2 (page 20). The following section is an evaluation of the ReTSE model based on this framework .

8.3.1 Inclusiveness

Inclusiveness is measured by safety [88]. The regression test selection technique is *safe* if the technique selects all the modification revealing test cases from the test suite of the Certified Program (C) for the Modified Program (M). The Rothermel and Harrold's framework claims that a regression test selection technique is not *safe* if it does not take into consideration the effects of new and deleted code. The inclusiveness of the regression test selection technique is the percentage given by the expression TCS / MR x (100) in cases where MR is not equal to 0. TCS is the number of test cases that are selected by the technique. MR is the number of modification revealing test cases which are taken from TS and execute the modified parts of the program. Inclusiveness is 100% in case where MR is equal to 0. The technique is *safe* if inclusiveness is 100%.

Table 8.2 shows the calculation of inclusiveness of the ReTSE model for each case study described in Chapter 7. The calculation of inclusiveness is based on the number of selected test cases which is divided into two parts: before and after the Optimisation Phase (RT & RTO). RT is a set of Regression Tests that are selected before the Optimisation Phase. The number of RT should be less than or equal to the number of MR. If the RT is equal to MR, then the ReTSE model is *safe*, while if the RT is less than MR then the model is less *safe*. RTO is a set of Optimised Regression Tests that are selected after the Optimisation Phase. The number of RTO should be less than or equal to the number of RT. The Optimisation Phase is designed to reduce redundant test cases having the same coverage and to identify the need of new test cases for the M.

Table 8.2: Summary of Inclusiveness Calculation

	Case 1 Change Statements	Case 2 Add Statements	Case 3 Delete Statements	Case 4 Add New Variables	Case 5 Delete Variables	Case 6 Change, Add & Delete Statement	Case 7 All Types of Modification (except Delete Statements)
No. of test cases in TS	6	6	6	6	6	6	6
No. of Modification revealing test cases (MR)	3	3	2	0	0	1	3
Before Optimisation Phase							
No. of test cases in RT	3	3	2	0	0	1	3
Inclusiveness (%) (RT/MR x 100)	100	100	100	100	100	100	100
After Optimisation Phase							
No. of test cases in RTO	1	1	1	0	0	1	1
Reduction of RTO (%) ((RT-RTO)/RT x 100)	66.67	66.67	50	-	-	-	66.67

The table shows that before the Optimisation Phase, the inclusiveness of the ReTSE model is 100% for all case studies. This means that the ReTSE model is *safe*. However, after the Optimisation Phase, the model has reduced a number of selected test cases in some cases because the model is also designed to tackle the issue of redundant test cases. For example, in Case 1, Case 2 and Case 7, the ReTSE model has reduced 66.67% of test cases in RT (before the Optimisation Phase) for RTO (after the Optimisation Phase). In Case 3, the reduction of test cases from RT to RTO is 50%. However, there is no reduction in Case 6 because

the RT and RTO hold the same test case. There is no selection of test cases in Case 4 and Case 5 because they have involved add and delete variables that will not affect other parts of the modified program.

Overall, the ReTSE model can be classified as a *safe* regression test selection technique. In addition, the ReTSE model has considered all types of basic modifications such as *change*, *delete* and *add* statements which are the main features of a *safe* technique. Moreover, the ReTSE model has also designed to tackle another two types of modifications which are delete and add variables in the M.

8.3.2 Precision

Precision measures the extent to which the regression test selection technique omits non-modification revealing test cases from the test suite of C to test M. The precision of a regression test selection technique is the percentage given by the expression TCE / NMR x (100) in cases where NMR is not equal to 0. NMR is the number of non-modification revealing test cases, and TCE is the number of test cases that are excluded by the technique. The non-modification revealing test cases are tests that execute the unchanged parts of the program. Precision is 100% if NMR = TCE or NMR = 0. The technique is *precise* if precision is 100%.

Table 8.3 shows the calculation of the precision of the ReTSE model for each case study described in Chapter 7. The calculation is based on the number of excluded test cases which is divided into two parts: before and after the Optimisation Phase (ET & ET_a). ET is a set of Excluded Tests that are excluded before

the Optimisation Phase. The number of ET should be less than or equal to the number of NMR. If the ET is equal to MNR, then the ReTSE model is *precise*. If the ET is less than NMR, then the model is less *precise*. ET_a is the number of test cases that are excluded from RT after the Optimisation Phase. The number of ET_a should be less than or equal to the number of RT.

Table 8.3: Summary of Precision Calculation

	Case 1 Change Statements	Case 2 Add Statements	Case 3 Delete Statements	Case 4 Add New Variables	Case 5 Delete Variables	Case 6 Change, Add & Delete Statement	Case 7 All Types of Modification (except Delete Statements)
No. of test cases in TS	6	6	6	6	6	6	6
No. of non-modification revealing test cases (NMR)	3	3	4	6	6	5	3
Before Optimisation Phase							
No. of excluded test cases before Optimisation (ET)	3	3	4	6	6	5	3
No. of test cases in RT	3	3	2	0	0	1	3
Precision (%) (ET/NMR x 100)	100	100	100	100	100	100	100
After Optimisation Phase							
No. of excluded test cases after Optimisation (ET_a)	2	2	1	-	-	-	2
ET_a (%) (ET_a/TS x 100)	66.67	66.67	50	-	-	-	66.67

The table shows that before the Optimisation Phase, the precision of the ReTSE model is always 100%. This means that the ReTSE model is *precise*.

However, after the Optimisation Phase, the model has increased the number of excluded test cases in some cases because the model is also designed to tackle the issue of redundant test cases. For instance, in Case 1, Case 2 and Case 7, the model has excluded 66.67% of test cases after the Optimisation Phase. In Case 3, the model has excluded 50% of test cases. However, in Case 4, Case 5 and Case 6, the exclusion is not relevant because there is only one test case or none in the RT.

Rothermel and Harrold [88] have claimed that any regression test selection technique that does not take into consideration the semantic differences between two programs is not *precise*. This is the case where both programs are syntactically different but semantically equivalent. Therefore, the ReTSE model can be classified as not *precise* because the model only considers syntactically different programs.

8.3.3 Efficiency

The efficiency of regression test selection techniques is measured in terms of their space and time requirements that lead to computational cost. There are a four factors that need to be considered when determining the efficiency of the technique [88]. The first factor is the phase of the lifecycle in which the technique performs its activities. The second factor is its ability to automate. The third factor is the extent to which the technique is capable of calculating information on program modifications. The fourth factor is the ability of the technique to handle cases in which the modified program contains multiple modifications.

As described in Chapter 5, the ReTSE model has four phases: (1) Program Analysis that includes Pretty Print and Slicing steps, (2) Comparison, (3) Exclusion, and (4) Optimisation. The Pretty Print step has time complexity linear in the size of original programs (OC & OM). In the Slicing step, the time complexity relatively depends on the number of variables in the programs. The time required for the Program Analysis Phase can be even less than that because the model has used an existing tool called *csurf* for slicing. The Comparison Phase has two parts. The first has time complexity linear in the number of slices in DS-C and DS-M, while the second has time complexity linear in the number of slices in D. The time required for the Comparison Phase can be less than estimated because the model also has used an existing tool called *diff* in that phase. Moreover, the comparison only concentrates on the specific modification parts that are obtained by the decomposition slicing technique in the model. The time complexity of the exclusion in phase (3) is linear to the number of test cases in the test suite. The optimisation in phase (4) has time complexity linear in the number of test cases in RT and the size of the union of slices in D.

8.3.4 Generality

The generality of a technique is its ability to function in a wide and practical range of situations. The model works for all types of modifications such as *add*, *delete* and *change* statements. The model can even handle cases of add and delete variables in the modified program. Although the model only concentrates on the intraprocedural program, it can be expanded to the interprocedural program in future. Although the technique has been evaluated for the C programming

language, in principal it can be easily extended to similar types of programming languages.

8.4 Comparison with Pythia Technique

The Pythia technique is used for this comparison because the ReTSE model has used the same program comparison tool called *diff*. However, in the Pythia model, the *diff* tool is used to compare two programs. On the other hand, the *diff* tool is used to compare two decomposition slices in the ReTSE model.

8.4.1 Applying the ReTSE Model using Power Program

Vokolos and Frankl [102] have used Power program to illustrate the Pythia, a textual differencing technique. The program aims to raise a floating point number to an integer power, using Dijkstra's algorithm. The program consists of two files: main.c and power.c. Each file contains one function. The old version of the main function (main.c) is shown in Figure 8.1 while the old version of the power function (power.c) is shown in Figure 8.2. Because the ReTSE model only concentrates on intraprocedural, it is assumed that the changes only occur in the power.c function. The new version of power function (power-v1.c) is shown in Figure 8.3. The old version is called Certified Program (C) and the new version is called Modified Program (M) in the ReTSE model.

Vokolos and Frankl used five test cases (TC_i) to test the Certified Program (C). The input and output values for each test cases are shown in Table 8.4. Test case TC_2 in Table 8.4 catches an error in the certified program. The TH_i of each TC_i for C (power.c program) is shown in Table 8.5. In their model, TH_i is called a basic block execution trace which is an execution trace of a test case based on a basic block concept. A basic block is a sequence of consecutive statements

```
     extern double power();
     extern double atof();
     extern int atoi();
     extern void printf();
     extern void exit();
     main (argc, argv)
     int argc;
     char *argv[];
     {
S1        double x;
S2        int n;
S3        if (argc !=3)
          {
S4           exit(0);
          }
S5        x = atof(argv[1]);
S6        n = atoi(argv[2]);
S7        printf("power(%.1f, %d)=\n", x, n);
S8        printf("%g\n\n", power(x, n));
     }
```

Figure 8.1: Certified Program (C)- main.c

with the property that control enters at the beginning statements and may leave only at the very last statement [102]. However, the TH_i in Table 8.5 is slightly different from their paper in order to consider a statement based used in the ReTSE model. The X symbol in Table 8.5 shows that the statement is executed for that TC_i. The - symbol is for statement not executed. These programs and information are used to illustrate the ReTSE model in this section.

8.4.1.1 Phase 1: Program Analysis

Step 1.1: Pretty Print

The Original Certified Program and the Original Modified Program of the

```
       double power(x, n)
       double x;
       register int n;
       {
S1        int recip, sgn;
S2        double y;
S3        if (n < 0)
          {
S4           recip = 1;
S5           n = -n;
          }
          else
          {
S6           recip = 0;
          }
S7        sgn = 1;
S8        if (x < 0.0e0)
          {
S9           x = -x;
          }
S10       for (y = 1.0e0; n > 0; --n)
          {
S11          while (n % 2 == 0)
             {
S12             x *= x;
S13             n /= 2;
             }
S14          y *= x;
          }
S15       if (recip != 0 && y != 0.0e0)
          {
S16          return (sgn*1.0e0/y);
          }
          else
          {
S17          return (sgn*y);
          }
       }
```

Figure 8.2: Certified Program (C)- power.c

```
       double power(x, n)
       double x;
       register int n;
       {
S1’       int recip, sgn;
S2’       double y;
S3’       if (n < 0)
          {
S4’          recip = 1;
S5’          n = -n;
          }
          else
          {
S6’          recip = 0;
          }
S7’       sgn = 1;
S8’       if (x < 0.0e0)
          {
S9’          x = -x;
S10’         if (n % 2 == 1)
             {
S11’            sgn = -1;
             }
          }
S12’      for (y = 1.0e0; n > 0; --n)
          {
S13’         while (n % 2 == 0)
             {
S14’            x *= x;
S15’            n /= 2;
             }
S16’         y *= x;
          }
S17’      if (recip != 0 && y != 0.0e0)
          {
S18’         return (sgn*1.0e0/y);
          }
          else
          {
S19’         return (sgn*y);
          }
       }
```

Figure 8.3: Modified Program (M)- power-v1.c

Table 8.4: Test Suite for Certified Program (Power Program)

Test Case (TC_i)	Input	Output
TC_1	-5.0^2	25
TC_2	-3.0^3	27
TC_3	2.0^0	1
TC_4	1.0^4	1
TC_5	0.0^{-1}	0

Table 8.5: Test History (TH_i) of TC_i for Certified Program (power.c)

Statement	TH_1	TH_2	TH_3	TH_4	TH_5
S1	X	X	X	X	X
S2	X	X	X	X	X
S3	X	X	X	X	X
S4	-	-	-	-	X
S5	-	-	-	-	X
S6	X	X	X	X	-
S7	X	X	X	X	X
S8	X	X	-	-	-
S9	X	X	-	-	-
S10	X	X	X	X	X
S11	X	X	-	X	X
S12	X	X	-	X	-
S13	X	X	-	X	-
S14	X	X	-	X	X
S15	X	X	X	X	X
S16	-	-	-	-	-
S17	X	X	X	X	X

power program are assumed to have gone through the Pretty Print Step. The outputs of this step are a Certified Program (C) and Modified Program (M) as shown in Figure 8.2 and Figure 8.3 respectively. The M has two additional new statements at S10' and S11' in order to tackle the problem raised in the Certified Program (C).

Step 1.2: Slicing

In the Slicing Step, both C and M are decomposed into decomposition slices corresponding to their variables in the programs. Therefore, both programs have five decomposition slices corresponding to five variables which are x, *recip*, n, *sgn* and y. The decomposition slices for C (DS-C) are shown in Figures 8.4 to 8.7 in part (a). The decomposition slices for M (DS-M) are shown in part (b) of the same figures. Decomposition slices for variables *sgn* and y are similar as shown in Figure 8.7. The output summary of the Slicing Step is given below:

- DS-C = {DS-C$_x$, DS-C$_{recip}$, DS-C$_n$, DS-C$_{sgn}$, DS-C$_y$}
- DS-M = {DS-M$_x$, DS-M$_{recip}$, DS-M$_n$, DS-M$_{sgn}$, DS-M$_y$}

8.4.1.2 Phase 2: Comparison

In the first part of the Comparison Phase, the decomposition slices in the DS-C are compared to the decomposition slices in the DS-M using the *diff* tool. There is no output produced from the *diff* tool for the comparison between DS-C$_x$ and DS-M$_x$ (Figure 8.4). Similar result was achieved in the comparison between DS-C$_{recip}$ and DS-M$_{recip}$ (Figure 8.5). Therefore, these decomposition slices are included in a set of pairs of Similar Decomposition Slice (S). An output is produced from the

```
   double power(x, n)
   double x;
   register int n;
   {
S3   if (n < 0)
     {
S5     n = -n;
     }
     else
S8   if (x < 0.0e0)
     {
S9     x = -x;
     }
S10  for(y=1.0e0; n>0; --n)
     {
S11    while (n % 2 == 0)
       {
S12      x *= x;
S13      n /= 2;
       }
S14    y *= x;
     }
   }
```

(a) DS-C$_x$

```
   double power(x, n)
   double x;
   register int n;
   {
S3'  if (n < 0)
     {
S5'    n = -n;
     }
     else
S8'  if (x < 0.0e0)
     {
S9'    x = -x;
     }
S12' for(y=1.0e0; n>0; --n)
     {
S13'   while (n % 2 == 0)
       {
S14'     x *= x;
S15'     n /= 2;
       }
S16'   y *= x;
     }
   }
```

(b) DS-M$_x$

Figure 8.4: Decomposition Slice for Variable x (Power Program)

```
  double power(x, n)
  double x;
  register int n;
  {
S1  int recip, sgn;
S3  if (n < 0)
    {
S4    recip = 1;
    }
    else
    {
S6    recip = 0;
    }
S15 if(recip!=0 && y!=0.0e0)
    else
  }
```

(a) DS-C$_{recip}$

```
   double power(x, n)
   double x;
   register int n;
   {
S1'  int recip, sgn;
S3'  if (n < 0)
     {
S4'    recip = 1;
     }
     else
     {
S6'    recip = 0;
     }
S17' if(recip!=0 && y!=0.0e0)
     else
   }
```

(b) DS-M$_{recip}$

Figure 8.5: Decomposition Slice for Variable *recip* (Power Program)

```
[L1] double power(x, n)
     double x;
     register int n;
     {
S3     if (n < 0)
       {
S5       n = -n;
       }
S10    for(y=1.0e0; n>0; --n)
[L10]  {
S11      while (n % 2 == 0)
         {
S13        n /= 2;
         }
       }
[L16]}
```

(a) DS-C_n

```
[L1] double power(x, n)
     double x;
     register int n;
     {
S3'    if (n < 0)
       {
S5'      n = -n;
       }
S8'    if (x<0.0e0)
[L10]  {
S10'     if(n%2 == 1)
       }
S12'   for(y=1.0e0; n>0; --n)
       {
S13'     while (n % 2 == 0)
         {
S15'       n /= 2;
         }
       }
[L20]}
```

(b) DS-M_n

Figure 8.6: Decomposition Slice for Variable n (Power Program)

diff tool for the comparison between DS-C_n and DS-M_n (Figure 8.6). Therefore, both decomposition slices are included in a set of pairs of Difference Decomposition Slice (D). The DS-C_{sgn} and DS-M_{sgn} (Figure 8.7) are also included in the D because the output is produced from the *diff* tool for this comparison. The DS-C_y and DS-M_y (Figure 8.7) are also included in the D because the *diff* tool also produces an output from this comparison. The output summary of the first part of the Comparison Phase is shown in Table 8.6.

The second part of the Comparison Phase is a more detailed comparison between DS-Cv_i and DS-Mv_i only if they are members of D. In this case, only the decomposition slices for variables *sgn* (DS-C_{sgn} and DS-M_{sgn}), n (DS-C_n and DS-M_n) and y (DS-C_y and DS-M_y) are involved in the second part of the comparison

```
   double power(x, n)
   double x;
   register int n;
   {
S1   int recip, sgn;
S2   double y;
S3   if (n < 0)
     {
S4     recip = 1;
S5     n = -n;
     }
     else
     {
S6     recip = 0;
     }
S7   sgn = 1;
S8   if (x < 0.0e0)
     {
S9     x = -x;
     }
S10  for(y=1.0e0; n>0; --n)
     {
S11    while (n % 2 == 0)
       {
S12      x *= x;
S13      n /= 2;
       }
S14    y *= x;
     }
S15  if(recip!=0 && y!=0.0e0)
     {
S16    return (sgn*1.0e0/y);
     }
     else
     {
S17    return (sgn*y);
     }
   }
```

(a) DS-C$_{sgn}$/DS-C$_y$

```
   double power(x, n)
   double x;
   register int n;
   {
S1'  int recip, sgn;
S2'  double y;
S3'  if (n < 0)
     {
S4'    recip = 1;
S5'    n = -n;
     }
     else
     {
S6'    recip = 0;
     }
S7'  sgn = 1;
S8'  if (x < 0.0e0)
     {
S9'    x = -x;
S10'   if (n % 2 == 1)
       {
S11'     sgn = -1;
       }
     }
S12' for(y=1.0e0; n>0; --n)
     {
S13'   while (n % 2 == 0)
       {
S14'     x *= x;
S15'     n /= 2;
       }
S16'   y *= x;
     }
S17' if(recip!=0 && y!=0.0e0)
     {
S18'   return (sgn*1.0e0/y);
     }
     else
     {
S19'   return (sgn*y);
     }
   }
```

(b) DS-M$_{sgn}$/DS-M$_y$

Figure 8.7: Decomposition Slice for Variable *sgn/y* (Power Program)

Table 8.6: Comparison Results between DS-Cv_i and DS-Mv_i (Power Program)

Set of	Member of Set
D	{(DS-C$_n$, DS-M$_n$), (DS-C$_{sgn}$, DS-M$_{sgn}$), (DS-C$_y$, DS-M$_y$)}
S	{(DS-C$_x$, DS-M$_x$), (DS-C$_{recip}$, DS-M$_{recip}$)}
S	{}
L	{}

because they are members of D (shown in the first part of the comparison). The comparison output using the *diff* tool is given below:

- Comparison between DS-C$_{sgn}$ and DS-M$_{sgn}$

```
19a20,23
>     if (n % 2 == 1)
>     {
>        sgn = -1;
>     }
```

- Comparison between DS-C$_n$ and DS-M$_n$

```
8a9,12
>     if (x < 0.0e0)
>     {
>        if (n % 2 == 1)
>     }
```

- Comparison between DS-C$_y$ and DS-M$_y$

```
19a20,23
>     if (n % 2 == 1)
>     {
>        sgn = -1;
>     }
```

In the comparison between DS-C$_{sgn}$ and DS-M$_{sgn}$ (Figure 8.7), the statement at line 19 ([L9]) from DS-C$_{sgn}$ is included in the set of Change Statement for

variable *sgn* (CS_{sgn}). Any statement in the range of lines 20 ([L20]) to 23 ([L23]) from DS-M_{sgn} is included in the set of Relevant Change Statement for variable *sgn* (RCS_{sgn}). Therefore, S9 from DS-C_{sgn} is included in the CS_{sgn} and statements S10' and S11' from DS-M_{sgn} are included in the RCS_{sgn}. Statement S9' is also included in the the RCS_{sgn} because it is located at the same branch of statement S10'. The same happens to the comparison between DS-C_y and DS-M_y (Figure 8.7) where statement S9 from DS-C_y is included in the CS_y and statements S9', S10' and S11' from DS-M_y are included in the RCS_y.

In the comparison between DS-C_n and DS-M_n (Figure 8.6), line 8 ([L8]) is not a statement but a close curly bracket (}). Therefore, the statement immediately after that symbol will be included in the set of Change Statement for variable *n* (CS_n). Any statement in the range of lines 9 ([L9]) to 12 ([L12]) from DS-M_n is included in the set of Relevant Change Statement for variable *n* (RCS_n). Therefore, statement S10 is included in the CS_n and statements S8' and S10' are included in the RCS_n. Then the CS is produced from the union of CS_{sgn}, CS_y and CS_n where the RCS is produced from the union of RCS_{sgn}, RCS_y and RCS_n. A summary of the second part of the Comparison Phase is given below:

$$\begin{aligned} CS &= CS_{sgn} \cup CS_y \cup CS_n \\ &= \{S9\} \cup \{S9\} \cup \{S10\} \\ &= \{S9, S10\} \end{aligned}$$

$$\begin{aligned} \text{RCS} &= \text{RCS}_{sgn} \cup \text{RCS}_{y} \cup \text{RCS}_{n} \\ &= \{\text{S9', S10', S11'}\} \cup \{\text{S9', S10', S11'}\} \cup \{\text{S8', S10'}\} \\ &= \{\text{S8', S9', S10', S11'}\} \end{aligned}$$

8.4.1.3 Phase 3: Exclusion

There are five test cases (TC_i) were used by Vokolos and Frankl for Certified Program (C) of power.c program as shown in Table 8.4. Their TH_i is shown in Table 8.5. Any TC_i where the CS is not subset of TH_i, will be included in the set of Excluded Test (ET). In this case, the CS which includes statements S9 and S10, is not subset of TH_3, TH_4 and TH_5. Therefore, TC_3, TC_4, and TC_5 will be included in the set of ET. The remaining test cases in Test Suite are included in the set of Regression Tests (RT). Therefore, TC_1 and TC_2 are included in the RT. A summary of the Exclusion Phase is given below:

$$\text{RT} = \{\text{TC}_1, \text{TC}_2\}$$

8.4.1.4 Phase 4: Optimisation

In the Optimisation Phase, all TC_i that are members of the RT will be executed onto UDS-M. In this case, there are only three DS-Mv_i members of D which are DS-M_{sgn}, DS-M_n and DS-M_y. The union of these decomposition slices (UDS-M) has produced the same program as M (power-v1.c) as shown in Figure 8.3 which includes statements S1', S2', S3', S4', S5', S6', S7', S8', S9', S10', S11', S12', S13', S14', S15', S16', S17', S18' and S19'. The RCS produced in the second part of the Comparison Phase is used here. The RCS includes statements S8', S9', S10' and S11'.

Test cases in RT, which are TC_1 and TC_2, are sequently executed onto the UDS-M. Firstly, TC_1 is executed onto UDS-M. The RTE_1 for TC_1 is S1', S2', S3', S6', S7', S8', S9', S10', S11', S12', S13', S14', S15', S16', S17' and S19'. The RTE_1 contains all members of RCS. That means the RCS receive full coverage by executed only the TC_1. The execution of test cases is stopped because the RCS has already achieved full coverage. This means it is enough to use only TC_1 as a regression test for the modified program. Therefore, TC_1 will be included in the RTO. The remaining test case TC_2 will be ignored for RTO. A summary of the Optimisation Phase is given below:

UDS-M = {S1',S2',S3',S4',S5',S6',S7',S8',S9',S10',S11',S12',S13',S14', S15',S16',S17',S18',S19'}

RCS = {S8',S9',S10',S11'}

TC_1

RTE_1 = {S1',S2',S3',S6',S7',S8',S9',S10',S11',S12',S13',S14', S15',S16',S17',S19'}

RCS-current = {S8', S9', S10', S11'}

RCS-coverage = {S8', S9', S10', S11'}

RTO = $\{TC_1\}$

RCS-full = YES

Therefore, the final output of the model for this case is given below:

RTO = $\{TC_1\}$

Request_New_Test_Cases = NO

8.4.2 Results Comparison

The Pythia has selected two test cases from an existing test suite to test a new version of the power program [102]. The same program has been applied to the ReTSE model. In the Exclusion Phase, the ReTSE model has selected the same two test cases as the Pythia technique. Moreover, after Optimisation Phase, the ReTSE model has selected only one test case to test a new version of the power program. This is because the model has identified that both test cases (produced in the Exclusion Phase) are redundant at the same coverage of a new version of the program. However, this type of a program can be classified as an unsuccessful case for the ReTSE model because it has computed a bigger decomposition slices for variables sgn and y. The slices are the same size as M. This issue will consume more time and cost for the ReTSE model compared to the Pythia.

8.5 Summary

This chapter has discussed the analysis and evaluation of the ReTSE model. The analysis done on the results obtained from case studies in the previous chapter. The ReTSE model has been evaluated based on the framework developed by Rothermel and Harrold. Finally, the ReTSE model has been applied to the program that has been used in the Pythia paper. The results of this analysis and evaluation show that the model is capable of producing a significant reduction of test cases for the Modified Program (M).

Chapter 9

Conclusions

9.1 Introduction

This chapter summarises and reviews the research. It includes the research background, the proposed model, its prototype, analysis and evaluation. It also discusses achievements of the research based on the criteria for success as defined in Chapter 1. Finally, suggestions are made for future directions.

9.2 Thesis Summary

The research in this thesis is about regression testing specifically on developing a regression test selection model using the decomposition slicing technique. Regression testing is the process of attempting to validate a *modified program*, a change to a previously tested version of the program which is called the *certified program*. It is also to ensure that the modifications of the program did not introduce any unexpected errors. An important issue in regression testing is how to reuse the

existing test suite for the *modified* program. One of the techniques to tackle this issue is regression test selection. Regression test selection technique attempts to reduce the time required to retest a *modified* program by selecting a subset of the existing test suite and retesting only the relevant parts of the *modified program*.

The proposed regression test selection model in this thesis has used decomposition slicing technique as a program analysis tool at an early phase of the model. The main objective of the decomposition slicing technique is to decompose a program directly into two parts: decomposition slice and complement. The decomposition slice is built for one variable, which is the union of slices taken at certain line numbers where the uses of that variable are located in the program. The complement is the sub-program that remains after decomposition slice is removed from the original program. Therefore, the decomposition slicing provides a technique that is capable of identifying the unchanged parts of the system.

The proposed model is called a Regression Test Selection by Exclusion (ReTSE) and has been defined in Chapter 5. The ReTSE model has four main phases. They are Program Analysis which includes Pretty Print and Slicing steps, Comparison, Exclusion and Optimisation phases. The Pretty Print Step is a process to standardise the layout of the program and make the Comparison Phase later in the model easier to perform. In the Slicing Step, both Certified Program (C) and Modified Program (M) have been sliced using the decomposition slicing technique. The number of the decomposition slices produced is dependent on the number of variables in the program. The Comparison Phase is divided into two parts. The first is a comparison between decomposition slices for the C and M

programs. This part determines the decomposition slices to included in S (set of pairs of Similar Decomposition Slices), D (set of pairs of Difference Decomposition Slices), N (set of New Decomposition Slices) or L (set of Delete Decomposition Slices). The second part is a more detailed comparison between the decomposition slices only for those that are members of the set of D by analysing the output from the comparison tool. This part produces a set of Change Statement (CS) and a set of Relevant Change Statement (RCS). In the Exclusion Phase, any test cases in the existing test suite have been excluded based on information from CS, test cases (TC) and its test histories (TH). The remaining test cases in test suite have been included in the set of Regression Tests (RT). Finally, in the Optimisation Phase, the number of test cases in RT has been reduced again based on information from D, N, RT and RCS. This phase specifically focuses on redundant test cases.

Currently, the prototype of the ReTSE model is not fully automated. However, some phases in the model have used existing tools. For instance, the Slicing Step in the Program Analysis Phase has used CodeSurfer (*csurf*) tool as assistant in order to produce decomposition slices for both C and M programs. In the Comparison Phase, the *diff* tool has been used intensively in order to compare decomposition slices from both programs. The other phases are carried out manually.

The validity of the ReTSE model is explored through the application of a number of case studies. This thesis has described the application of seven case studies in order to analyse and evaluate the model. Five of them correspond

to the five types of modifications that can be applied to a program. These are change statements, add statements, delete statements, add variables and delete variables. Another two case studies are presented for multiple types of modification at a time. The case studies have shown that reductions in the number of test cases can be achieved. For instance, in the Exclusion Phase, the ReTSE model has reduced more than 50% of test cases in the test suite. In the Optimisation Phase, the model once again has reduced at least 50% because of redundant test cases. At some points, the model has reduced 100% of test cases in test suite. This is because the model has involved only in add or delete variables. These cases can be detected by using the decomposition slicing technique as shown in Case 4 (page 115) and Case 5 (page 122) in Chapter 7.

The evaluation of the model is divided into two parts as described in Chapter 8. The first part is an evaluation based on the Rothermel and Harrold's evaluation framework [88]. The framework has four categories which are inclusiveness, precision, efficiency and generality. The results from the case studies show that before the Optimisation Phase, the inclusiveness and precision of the ReTSE model are always 100%. This means that the ReTSE model is *safe* and *precise*. Moreover, after the Optimisation Phase, the number of test cases in RT has reduced in some cases such as Case 1 (page 97), Case 2 (page 98), Case 3 (page 107) and Case 7 (page 137). This is because the model is also designed to tackle the issue of redundant test cases. The efficiency of the ReTSE model is based on the time complexity of each phase which is linear to it input. In term of generality, the model works for all types of modifications such as add, delete and change statements. Moreover, the model can handle cases of add and delete

variables in *modified* program.

The second part of the evaluation in Section 8.4 (Chapter 8) is based on the comparison between the ReTSE model and the Pythia technique. In the paper written by Vokolos and Harrold [102], the Pythia technique has selected two test cases from an existing test suite to test a new version of the power program. The same program has been applied with the ReTSE model. In the Exclusion Phase, the ReTSE model has selected the same two test cases as the Pythia technique. Moreover, the ReTSE model has finally selected only one test case to test a new version of the power program which is called Modified Program (M) in the ReTSE model. This is because the model has identified that these test cases are redundant and have the same coverage of the M program.

The results of these analysis and evaluation show that the ReTSE model is capable of producing a significant reduction of test cases for M.

9.3 Criteria for Success

The criteria for success of the research in this thesis have been presented in Chapter 1. This section discusses the achievements of these criteria. These achievements are as follows.

1. The development of a new regression test selection model

   The developed model in this thesis is called Regression Test Selection by Exclusion (ReTSE). The model has four main phases as described in Chapter 5. The first phase is Program Analysis which includes Pretty Print and

Slicing steps. This phase is discussed in Section 5.2.1 (page 58). The second phase is called Comparison. This phase is divided into two parts as described in Section 5.2.2 (page 62). The third phase is called Exclusion as described in Section 5.2.3 (page 67). This phase has selected relevant test cases for the Modified Program (M). The last phase is called Optimisation as discussed in Section 5.2.4 (page 68). This phase has excluded redundant test cases that have been selected in the Exclusion Phase.

2. The implementation of the new model

   This study has implemented part of the ReTSE model into a prototype tool. Currently, the prototype of the model is not fully automated as described in Chapter 6. However, some phases of the model have used suitable existing tools. For instance, the Slicing Step in the Program Analysis Phase has used CodeSurfer (*csurf*) tool in order to produce decomposition slices for Certified Program (C) and Modified Program (M) (see Section 6.2.1.2, page 89). The Comparison Phase has used the *diff* tool in order to compare between decomposition slices from C and M (see Section 6.2.2, page 93). The other phases work manually and all phases are not integrated with each other. However, some suggestions for further implementation are described in Chapter 6.

3. An analysis of the new model

   This thesis has also presented the analysis of the ReTSE model. It begins with the application of case studies that represent all types of modifications such as change statements, add statements, delete statements, add variables and delete variables. Those case studies are described in Chapter

7. Section 7.2 (page 97) has presented five case studies that have used one type of modification at a time. Section 7.3 (page 127) has discussed two case studies that have a combination of modification types. The analysis of these case studies has provided in Chapter 8, specifically in Section 8.2 (page 149).

4. An evaluation of the new model

   In Chapter 8, the ReTSE model has been evaluated based on two parts. The first evaluation is based on an existing evaluation framework proposed by Rothermel and Harrold. This evaluation is described in Section 8.3 (page 151). The second evaluation is a comparison between the ReTSE model and the Pythia technique, and the evaluation is described in Section 8.4 (page 159).

9.4 Future Directions

Although the proposed model presented in this thesis has considerably achieved the intended goals, there are many potential extensions that can be enhanced in the future. These extensions are as follows.

1. Fully implemented and integrated

   Some of the phases in the ReTSE model have used existing tools, while others are done manually. Therefore, in the future, the prototype of the model can be expanded to make it fully automated. Then, all phases in the model can be integrated as a complete tool.

2. Designing the new requested test cases

   In the ReTSE model, there are two outputs of the Optimisation Phase. They are a set of optimised regression tests (RTO) and a request for new test cases. Currently, the model only decides whether new test cases are needed or not. However, the process of designing the additional new test cases is beyond the scope of the ReTSE model. Hence, this process can be extended in the future.

3. Running the selected and new test cases on the modified parts only

   If the designing of new test cases is adopted in the ReTSE model, then the model can be expanded to run all relevant test cases on the modified parts only. The relevant test cases involve a set of Optimised Regression Tests (RTO) that are produced from the Optimisation Phase and a set of test cases that are produced from the designing new test cases. The modified parts include a combination of the DS-Mv_i where DS-Mv_i is a member of D and N (refer to Chapter 5, Section 5.2.2, page 62).

4. Improving generality of the model

   Currently, the ReTSE model only works for a single module or function of C programs and this concept is called intraprocedural. The model can be expanded to make it more generic by incorporating interprocedural concepts. This can be realised by doing some programming using Schema, a scripting language that exists in csurf tool in order to produce decomposition slicing automatically. The ReTSE model also can be applied to other programming language.

5. Evaluating the model using SIR larger programs

At this moment, the thesis has used small programs in order to analyse and evaluate the ReTSE model. Those small programs are discussed in case studies in Chapter 7. Therefore, the further research can evaluate the model for larger programs. This can be achieved with the prerequisites of suggestions 1 and 4. The larger programs can be taken from Software-artifact Infrastructure Repository (SIR) [27] that provides a lot of sources for program analysis and software testing.

9.5 Summary

This thesis has discussed a research in the area of regression testing. Specifically, this research focuses on developing a model for regression test selection by exclusion using the decomposition slicing technique called ReTSE. The case studies have shown that the model has given a significant reduction in the number of test cases that need to be run after changes have been made. Evidently, the ReTSE model offer significant opportunity for enhancement and improvement for future research.

The PhD process is an iterative process that requires a step by step progression from simple ideas to more complex abstract notions. In this respect the development of the ReTSE model took a while to develop and specify and was done at the expense of developing a more technical tool for the implementation. Working through simple examples of C code also contributed to understanding how the model should be constructed.

References

[1] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, 1990. 1

[2] IEEE Standard for Software Maintenance. *IEEE Std 1219-1993*, Jun 1993. 1

[3] IEEE Standard for Software Maintenance. *IEEE Std 1219-1998*, Oct 1998. ix, 15, 19

[4] ISO/IEC Standard for Software Maintenance. *ISO/IEC Std 14764:1999*, 1999. 2

[5] Grammatech, inc,. *http://www.grammatech.com/products/codesurfer/*, Last accessed 31 Nov 2011. 89

[6] Hiralal Agrawal and Joseph Robert Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'90)*, pages 246–256, 1990. ix, 38, 45, 46, 53

[7] Hiralal Agrawal, Joseph Robert Horgan, Edward W. Krauser, and Saul London. Incremental regression testing. In *Proceedings of the International Conference on Software Maintenance (ICSM'93)*, pages 348–357, 1993. 29

[8] Martyn A.Ould and Charles Unwin. *Testing in Software Development*. The British Computer Society, 1986. ix, 8, 9

[9] Janvi Badlaney, Rohit Ghatol, and Romit Jadhwani. An introduction to data-flow testing. TR 22, North Carolina State University, August 2006. 12

[10] Ghinwa Baradhi and Nashat Mansour. A comparative study of five regression testing algorithms. In *Proceedings of the Australian Software Engineering Conference (ASWEC'97)*, pages 174–, 1997. 2, 22, 30

[11] Keith H. Bennett and Václav Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the International Conference on Software Engineering (ICSE'00)*, pages 73–87, 2000. 1

[12] Jean-Francois Bergeretti and Bernard A. Carré. Information-flow and data-flow analysis of while-programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(1):37–61, 1985. 38, 47

[13] Antonia Bertolino. Software testing research: Achievements, challenges, dreams. In *Proceedings of the Future of Software Engineering Symposium(FOSE'07)*, pages 85–103. IEEE Computer Society, 2007. 1

[14] David Binkley. The application of program slicing to regression testing. *Information & Software Technology*, 40(11-12):583–594, 1998. 2, 23, 29

[15] David Binkley and Keith Brian Gallagher. Program slicing. *Advances in Computers*, 43:1–50, 1996. 38, 42

[16] David Binkley, Mark Harman, L. Ross Raszewski, and Christopher Smith. An empirical study of amorphous slicing as a program comprehension support tool. In *Proceedings of the 8th International Workshop on Program Comprehension*, pages 161–170, 2000. 38, 53, 54

[17] Pierre Bourque, Robert Dupuis, Alain Abran, James W.Moore, and Leonard Tripp. The guide to the software engineering body of knowledge 2004 version. *IEEE Software*, 16(6):35–44, 1999. 7

[18] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Information & Software Technology*, 40(11-12):595–607, 1998. 38, 47, 53

[19] Jim Kingdon David Ingamells Carlo Wood, Joseph Arceneaux. Indent. *http://www.gnu.org/software/indent/manual/*, Last accessed 2 November 2011. 89

[20] Lin Chen, Ziyuan Wang, Lei Xu, Hongmin Lu, and Baowen Xu. Test case prioritization for web service regression testing. In *Proceedings of the 5th IEEE International Symposium on Service Oriented System Engineering (SOSE'10)*, pages 173 –178, june 2010. 23

[21] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. Testtube: A system for selective regression testing. In *Proceedings of the International Conference on Software Engineering (ICSE'94)*, pages 211–220, 1994. 2, 22, 23, 28, 32

[22] Pavan Kumar Chittimalli and Mary Jean Harrold. Regression test selection on system requirements. In *Proceedings of the 1st India Software Engineer-*

ing Conference (ISEC'08), ISEC '08, pages 87–96, New York, NY, USA, 2008. ACM. 2, 22

[23] Bill Councill and George T. Heineman. Definition of software component and its elements. In *Component-Based Software Engineering*, pages 5–19. Addison-Wesley, 2001. 13

[24] Sebastian Danicic, Andrea De Lucia, and Mark Harman. Building executable union slices using conditioned slicing. In *Proceedings of the International Workshop on Program Comprehension (IWPC'04)*, pages 89–99, 2004. 47, 73

[25] Mohammed Daoudi, Lahcen Ouarbya, John Howroyd, Sebastian Danicic, Mark Harman, Chris Fox, and Martin P. Ward. Consus: A scalable approach to conditioned slicing. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE'02)*, pages 109–118, 2002. 47

[26] Paul Eggert David MacKenzie and Richard Stallman. Comparing and merging files. *http://www.gnu.org/software/diffutils/*, Last accessed 2 November 2011. 93

[27] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005. 94, 181

[28] Hyunsook Do and Gregg Rothermel. An empirical study of regression testing techniques incorporating context and lifetime factors and improved cost-

benefit models. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2, 22

[29] Chen Duanzhi. Program slicing. In *Preceedings of the International Forum on Information Technology and Applications*, pages 15–18, 2010. 38, 53

[30] Elena Dubrova. Structural testing based on minimum kernels. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'05)*, pages 1168–1173. IEEE Computer Society, 2005. 11, 12

[31] S. Elbaum, A.G. Malishevsky, and G. Rothermel. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159 –182, feb 2002. 23

[32] Sebastian G. Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'00)*, pages 102–112, 2000. 23

[33] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics*. International Thomson Publishing, 1997. 83

[34] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987. 40

[35] István Forgács, Ákos Hajnal, and Éva Takács. Regression slicing and its use in regression testing. In *Proceedings of the IEEE International Com-*

puter Software and Applications Conference (COMPSAC'98), pages 464–469, 1998. 23

[36] P.G. Frankl and E.J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, 1988. 12

[37] Keith Gallagher and David Binkley. An empirical study of computation equivalence as determined by decomposition slice equivalence. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE'03)*, pages 316–322, 2003. 50

[38] Keith Gallagher and David Binkley. Program slicing. In *Proceedings of the Frontiers of Software Maintenance*, pages 58–67, 2008. 38, 53

[39] Keith Gallagher, David Binkley, and Mark Harman. Stop-list slicing. In *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'06)*, pages 11–20, 2006. 38, 48

[40] Keith Gallagher, Tracy Hall, and Sue Black. Reducing regression test size by exclusion. In *Proceedings of the International Conference on Software Maintenance (ICSM'07)*, pages 154–163, 2007. 30, 37, 51

[41] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991. ix, 23, 38, 48, 49, 50, 51, 52, 53, 54

[42] Jerry Gao, Deepa Gopinathan, Quan Mai, and Jingsha He. A systematic regression testing method and tool for software components. In *Proceedings*

of the IEEE International Computer Software and Applications Conference (COMPSAC'06), pages 455–466, 2006. 35

[43] R.P. Gorthi, A. Pasala, K.K.P. Chanduka, and B. Leong. Specification-based approach to select regression test suite to validate changed software. In *Proceedings of the 15th Asia-Pacific Software Engineering Conference (APSEC'08)*, pages 153 –160, dec. 2008. 2, 22

[44] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Transactions on Software Engineering and Methodology*, 10:188–197, 2001. 2, 22

[45] R. Gupta, M.J. Harrold, and M.L. Soffa. An approach to regression testing using slicing. In *Proceedings of the International Conference on Software Maintenance (ICSM'92)*, pages 299–308, Nov 1992. 30

[46] Mark Harman and Robert M. Hierons. An overview of program slicing. *Software Focus*, 2(3):85–92, 2001. 53

[47] Mark Harman, Robert M. Hierons, Chris Fox, Sebastian Danicic, and John Howroyd. Pre/post conditioned slicing. In *Proceedings of the International Conference on Software Maintenance (ICSM'01)*, pages 138–147, 2001. ix, 47, 48

[48] Mary Jean Harrold. Testing evolving software: Current practice and future promise. In *Proceedings of the 1st India Software Engineering Conference (ISEC'08)*, pages 3–4. ACM, 2008. 1

[49] Mary Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2(3):270–285, 1993. 22

[50] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, Alessandro Orso, Maikel Pennings, Saurabh Sinha, S. Alexander Spoon, and Ashish Gujarathi. Regression test selection for java software. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'01)*, pages 312–326, 2001. 33, 34

[51] Mary Jean Harrold, David Rosenblum, Gregg Rothermel, and Elaine Weyuker. Empirical studies of a prediction model for regression test selection. *IEEE Transactions on Software Engineering*, 27, 2001. 2, 22

[52] M.J. Harrold and P. Kolte. Combat: A compiler based data flow testing system. In *Proceedings of the Pacific Northwest Quality Conference*, pages 311–323, 1992. 22

[53] J. Hartmann and D.J. Robson. Revalidation during the software maintenance phase. In *Proceedings of the Conference on Software Maintenance*, pages 70 –80, oct 1989. 2, 22

[54] J. Hartmann and D.J. Robson. Retest- development of a selective revalidation prototype environment for use in software maintenance. In *Proceedings of the 23rd Annual Hawaii International Conference on Systems Sciences*, volume ii, pages 92–101 vol.2, jan 1990. 2, 22

[55] Jean Hartmann and David J. Robson. Techniques for selective revalidation. *IEEE Software*, 7(1):31–36, 1990. 2, 22

[56] Bill Hetzel. *The Complete Guide to Software Testing (2nd ed.)*. QED Information Sciences, 1988. 1, 7

[57] Robert M. Hierons, Mark Harman, Chris Fox, Lahcen Ouarbya, and Mohammed Daoudi. Conditioned slicing supports partition testing. *Software Testing, Verification and Reliability*, 12(1):23–28, 2002. 47, 73

[58] Hyoung Seok Hong, Insup Lee, and Oleg Sokolsky. Abstract slicing: a new approach to program slicing based on abstract interpretation and model checking. In *Proceedings of the Fifth International Workshop on Source Code Analysis and Manipulation*, pages 25–34, 2005. 38, 53

[59] Susan Horwitz, Thomas W. Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, 1990. ix, 38, 40, 41, 42, 43, 47

[60] Hwa-You Hsu and Alessandro Orso. Mints: A general framework and tool for supporting test-suite minimization. In *Proceedings of the IEEE 31st International Conference on Software Engineering (ICSE'09)*, pages 419 –429, may 2009. 22, 23

[61] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software engineering (ICSE'94)*, pages 191–200, 1994. 25

[62] Takashi Ishio, Shinji Kusumoto, and Katsuro Inoue. Program slicing tool for effective software evolution using aspect-oriented technique. In *Proceedings*

of the Sixth International Workshop on Principles of Software Evolution, pages 3–12, 2003. 53

[63] D. Jeffrey and N. Gupta. Test case prioritization using relevant slices. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 411 –420, sept. 2006. 23

[64] Glenford J.Myers. *The Art of Software Testing*. John Wiley, 1979. 1, 7

[65] Glenford J.Myers, Badgett, Todd M. Thomas, and Crey Sandler. *The Art of Software Testing, Second Edition*. John Wiley and Sons, Inc., 2004. 1

[66] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach*. CRC Press, 1995. 10, 12

[67] Wang Jun, Zhuang Yan, and Jianyun Chen. Test case prioritization technique based on genetic algorithm. In *Proceedings of the 2011 International Conference on Internet Computing Information Services (ICICIS'11)*, pages 173 –175, sept. 2011. 23

[68] N. Kaushik, M. Salehie, L. Tahvildari, Sen Li, and M. Moore. Dynamic prioritization in regression testing. In *Proceedings of the IEEE 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'11)*, pages 135 –138, march 2011. 23

[69] Bogdan Korel and Janusz W. Laski. Dynamic program slicing. *Information Processing Letters*, 29(3):155–163, 1988. 38, 44, 53

[70] Phillip A. Laplante. *What Every Engineer Should Know About Software Engineering*. CRC Press, 2007. 1

[71] Yuejian Li and Nancy J. wahl. An overview of regression testing. *ACM SIGSOFT Software Engineering Notes*, 24(1):69–73, 1999. 19

[72] Zheng Li, Mark Harman, and Robert M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, 2007. 23

[73] Feng Lin, M. Ruth, and Shengru Tu. Applying safe regression test selection techniques to java web services. In *Proceedings of the International Conference on Next Generation Web Services Practices (NWeSP'06)*, pages 133–142, Sept. 2006. 34

[74] Andrea De Lucia. Program slicing: Methods and applications. In *Proceedings of the First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 142–149, 2001. 38, 53

[75] Zengkai Ma and Jianjun Zhao. Test case prioritization based on analysis of program structure. In *Proceedings of the 15th Asia-Pacific Software Engineering Conference (APSEC'08)*, pages 471 –478, dec. 2008. 23

[76] Christoph Malz and Peter Ghner. Agent-based test case prioritization. In *Proceedings of the IEEE 4th International Conference on Software Testing, Verification and Validation Workshops (ICSTW'11)*, pages 149 –152, march 2011. 23

[77] Amir Ngah and Keith Gallagher. Regression test selection by exclusion using decomposition slicing. In *Proceedings of the Doctoral Symposium for the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 23–24, 2009. 3

[78] S.C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, 1988. 12

[79] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 241–251, 2004. 32

[80] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the 1st ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, 1984. 40

[81] William Perry. *Effective Methods for Software Testing*. John Wiley, 1995. 8

[82] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, UK, 2010. 1

[83] Bo Qu, Changhai Nie, and Baowen Xu. Test case prioritization for multiple processing queues. In *Proceedings of the International Symposium on Information Science and Engineering (ISISE'08)*, volume 2, pages 646 –649, dec. 2008. 23

[84] Bo Qu, Changhai Nie, Baowen Xu, and Xiao fang Zhang. Test case prioritization for black box testing. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC'07)*, volume 1, pages 465 –474, july 2007. 23

[85] S. Rapps and E.J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, 1985. 12

[86] Juergen Rilling and Bhaskar Karanth. A hybrid program slicing framework. In *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 12–23, 2001. 38

[87] Marc Roper. *Software Testing*. McGraw Hill, 1994. 1, 8, 12

[88] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996. 3, 4, 20, 27, 28, 30, 148, 151, 152, 156, 176

[89] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering Methodology (TOSEM)*, 6(2):173–210, 1997. 2, 19, 21, 22, 23, 24, 25, 32, 33, 34, 35

[90] Gregg Rothermel and Mary Jean Harrold. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering*, 24:401–419, 1998. 2, 22, 25

[91] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *Proceedings of the International Conference on Software Maintenance (ICSM'99)*, pages 179–188, 1999. 23

[92] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, 2001. 21, 23

[93] Michael Ruth, Sehun Oh, Adam Loup, Brian Horton, Olin Gallet, Marcel Mata, and Shengru Tu. Towards automatic regression test selection for web services. In *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC'07)*, volume 2, pages 729 –736, july 2007. 35

[94] Michael Ruth and Shengru Tu. A safe regression test selection technique for web services. In *Proceedings of the International Conference on Internet and Web Applications and Services (ICIW'07)*, page 47, 2007. 35

[95] Goutam Kumar Saha. Understanding software testing concepts. *Ubiquity*, 9(6):1–1, 2008. 1, 7, 8

[96] N. Sasirekha, A. Edwin Robert, and M. Hemalatha. Program slicing techniques and its applications. *International Journal of Software Engineering and Applications (IJSEA)*, 2(3), 2011. 38, 53

[97] Ian Sommerville. *Software Engineering(7th ed.)*. Addison Wesley, 2004. ix, 1, 8, 13, 14

[98] Hema Srikanth, Laurie Williams, and Jason Osborne. System test case prioritization of new and regression test cases. In *Proceedings of the International Symposium on Empirical Software Engineering (ISESE'05)*, pages 64–73, 2005. 23

[99] Abbas Tarhini, Zahi Ismail, and Nashat Mansour. Regression testing web applications. In *Proceedings of International Conference on Advanced Computer Theory and Engineering (ICACTE'08)*, 2008. 34

[100] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995. ix, 39, 41, 44

[101] S.R. Vergilio, J.C. Maldonado, and M. Jino. Constraint based selection of test sets to satisfy structural software testing criteria. In *Proceedings of the International Conference of the Chilean Computer Science Society*, volume 0, page 256, 1997. 10

[102] F. I. Vokolos and P. G. Frankl. Pythia: a regression test selection tool based on textual differencing. In *Proceedings of the Internatinal Conference on Reliability, Quality and Safety of Software-intensive Systems(ENCRESS'97)*, pages 3–21. Chapman & Hall, Ltd., 1997. 2, 19, 22, 23, 25, 27, 31, 32, 148, 159, 160, 172, 177

[103] Filippos I. Vokolos and Phyllis G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Proceedings of the International Conference on Software Maintenance (ICSM'98)*, pages 44–53, 1998. 2, 22, 23, 27, 28

[104] Mark Weiser. Program slicing. In *Proceedings of the International Conference on Software Engineering (ICSE'81)*, pages 439–449, 1981. 29, 38, 42, 53

[105] M.D. Weiser, J.D. Gannon, and P.R. McMullin. Comparison of structural test coverage metrics. *IEEE Software*, 2(2):80–85, 1985. 10, 12

[106] Ye Wu, Mei-Hwa Chen, and H.M. Kao. Regression testing on object-oriented programs. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, pages 270–279, 1999. 33

[107] Lei Xu, Baowen Xu, Zhenqiang Chen, Jixiang Jiang, and Huowang Chen. Regression testing for web applications based on slicing. In *Proceedings of the IEEE International Computer Software and Applications Conference (COMPSAC'03)*, pages 652–656, 2003. 23

[108] Damiano Zanardini. The semantics of abstract program slicing. In *Proceedings of the Eighth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 89–98, 2008. 38

[109] Xiaofang Zhang, Changhai Nie, Baowen Xu, and Bo Qu. Test case prioritization based on varying testing requirement priorities and test case costs. In *Proceedings of the 7th International Conference on Quality Software (QSIC'07)*, pages 15 –24, oct. 2007. 23

[110] Jiang Zheng. In regression testing selection when source code is not available. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*. 2, 22